NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

FACILITATING RICH ACOUSTICAL ENVIRONMENTS IN VIRTUAL WORLDS

by

Kenneth J. Hoag Sr.

September 1998

Thesis Advisor:	Rudolph Darken
Second Reader:	Russell Storms

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 1998	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE **Facilitating Rich Acoustical Environments in Virtual Worlds**	**5. FUNDING NUMBERS**
6. AUTHOR(S) Hoag Sr., Kenneth J.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

The visual aspect of virtual environments has advanced at a rapid pace. The audio aspect, however, has not kept pace. Current methods of building virtual models do not address the graphical and audio aspects in an integrated fashion. Furthermore, graphical programming tools have not addressed sound in a satisfactory manner.

As proof of concept, a modeling tool was developed to allow a user to build both the visual and the auditory environment simultaneously. A rendering application was developed that would display and browse a graphical environment, an audio environment, or a complete graphical/audio environment.

This thesis demonstrates that building both the auditory and the visual geometry simultaneously allows for rapid, easy development of both the visual and the auditory environment. Enhancements and recommendations to current software technologies and modeling languages are introduced. New models to represent audio are introduced.

14. SUBJECT TERMS Virtual Audio, Virtual Environment, 3-D Audio, Spatialized Sound, Audio Environment	**15. NUMBER OF PAGES** 97
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 *Standard Form 298 (Rev. 2-89)* Prescribed by ANSI Std. 239-18

FACILITATING RICH ACOUSTICAL ENVIRONMENTS IN VIRTUAL WORLDS

Kenneth J. Hoag Sr.
Captain, United States Marine Corps
B.S., University of Southwestern Louisiana, 1986

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1998

Author: ______________________________
Kenneth J. Hoag Sr.

Approved by: ______________________________
Rudolph Darken, Thesis Advisor

____
Russell Storms, Second Reader

____
Dan Boger, Chair
Department of Computer Science

ABSTRACT

The visual aspect of virtual environments has advanced at a rapid pace. The audio aspect, however, has not kept pace. Current methods of building virtual models do not address the graphical and audio aspects in an integrated fashion. Furthermore, graphical programming tools have not addressed sound in a satisfactory manner.

As proof of concept, a modeling tool was developed to allow a user to build both the visual and the auditory environment simultaneously. A rendering application was developed that would display and browse a graphical environment, an audio environment, or a complete graphical/audio environment.

This thesis demonstrates that building both the auditory and the visual geometry simultaneously allows for rapid, easy development of both the visual and the auditory environment. Enhancements and recommendations to current software technologies and modeling languages are introduced. New models to represent audio are introduced.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost I would like to thank my wife, Rachel, for the support she has shown and the understanding she has displayed. For half of my life, you have been there for me. Thanks for tolerating all of my pet projects, of which this thesis is the closest to being completed. Thanks for asking me if I was putting in more work than the instructor really required, because often I was too involved to see that I was overshooting the requirements.

Second, I would like to thank my children, Kayleigh, Jordan, and Jamey. You helped me on projects when I asked, rode your bikes with me so that I would have to run faster, and helped me to keep my thinking fresh.

Dr. Darken, thanks for all of the help. I have seldom worked for anyone that would tell me what they wanted, then let me go about accomplishing it. The non-micromanagement was very productive. Thanks for letting me go out and experiment with a little bit of everything. Can you believe, I actually stayed with a topic for more than a few weeks?

Thanks to Major Russell Storms for being my second reader and one of the only people around that could really get down into the weeds when discussing sound. If you had not started working on three-dimensional sound at NPS, who knows what I would have worked on.

I'd also like to thank Scott Foster, Chief Technical Officer at Aureal Semiconductor. Thanks for the afternoon you spent giving me a different perspective on sound in general. I hope you don't mind my using your three state model.

The NPSNRG has many bright people, and they are always willing to help. Thanks to Kent Watsen for explaining things like the APP-CULL-DRAW process, and scene graphs, and how many times did I ask you about them anyway? Howard Abrams, your explanation of scene graph traversal was actually the turning point in my understanding graphics programming. Thanks to John Locke for all of the models that you built for us and for the teaching me the joy of debunking conspiracy theory. Special thanks to Rosalie Johnson and Jimmy Liberato, two of the most patient and competent support personnel I can imagine.

I. INTRODUCTION

A. MOTIVATION

Sound is a very important part of our perception of the world. The eyes perceive a great amount of information; however, they can only take in information within their field of view. If something happens behind, above, or below, or if we are visually concentrating on a single subject, our ears tell us when we need to look in a different direction to experience different visual information; the visual and aural perception systems are tightly integrated. At times, as with the sound of a gunshot or the screech of tires, sound can evoke a reaction with or without visual stimulation.

The visual aspect of virtual reality has matured to the point of reasonable believability. Images can jump out in three dimensions. As a user's point of view changes, the scene changes. As the environment changes, the user's view changes. As the sun sets, or a tree falls, or an enemy assumes the prone position to ready itself for a hasty ambush on a patrol, the visual information presented to the user's eyes changes appropriately. All of this visual stimulation is informative to the user of the virtual world, but is it enough? If the sun sets behind the user's back, how would the user know? The user might notice a decrease in overall brightness, may recognize a reddish-orange sky, or might even see shadows elongating. If a tree falls behind the user, how would the user know? Visually, it would not be perceived unless it fell into the field of view. If the enemy correctly sets that ambush, how would it be discovered? If they have prepared their ambush properly, the user would not notice them until they began firing, and even then, it would not be visually noticed if the gunfire came from the behind the user. The first scenario presents an example where visual may be all that is necessary. The second and third scenarios, however, cry out for sound.

Sound can become important in a virtual environment (VE) for many reasons. When an architect designs a building, a virtual mock-up of the building provides the architect, and the architect's client, an opportunity to walk through the building before it is built. When done in an immersive manner using a head-mounted-display (HMD), they can see how the walkways look, if the hallways are large enough, what the lighting will look like, if the furniture will fit, etc. If the user could also hear the new building, necessary design changes could be discovered before construction begins. Much information about size and scale is best captured by sound.

The concept of "hearing" a room also applies to people wanting to re-decorate or redesign a room. An interior decorator could simply build a three-dimensional replica of the room and apply appropriate textures to the walls. Carpeting, curtains, wall-coverings, and other materials could be placed into the room. The decorator could then experiment with different types of lighting and different arrangements of objects. The visual aspects of these changes would be readily apparent. If the room also contains a virtual stereo, public address system, piano, or even a modeled orator, the effects of the new decorations could be heard as well as seen.

An acoustical engineer may also want to use a system such as that described above. The acoustical engineer may want better or more accurate replication of the sound of the room, and that engineer would probably desire a method of developing a room that seamlessly integrated the construction of the visual space into the construction of the aural space.

Recent work by Major Russell Storms at the Naval Postgraduate School (NPS) has shown that the quality of sound has a direct effect on the perception of the graphics being displayed in a virtual environment. [STOR98] Other recent work by Major John Lawson at NPS shows that sound is strongly correlated to the feeling of immersion felt by a user of a virtual environment. [LAWS98]

B. RESEARCH OBJECTIVES

If we accept the above as true, we must agree that sound is a necessary part of a virtual environment. The next question is, "Why is sound not prevalent in virtual environments?". Sound is not prevalent in virtual environments because it is difficult to implement, and it is **especially** difficult to implement correctly. Sound support for virtual environments, however, is lacking in both the ability to generate sounds as well as the ability to model/develop the aural virtual environment. This thesis is concerned with making the integration of sound into virtual environments easier and more accurate. A modeler should be able to implement sound and audio into a model without needing expertise as both a geometric modeler and an acoustic modeler. A programmer should be able to implement the graphical and audio portions of a browser or modeler without making redundant calls to the graphical and the audio hardware. The programmer should not even need to know what type of audio hardware is present. Just as a graphical programmer should not need to know if the graphics are rendered on a PC or an SGI, the programmer should also not concern himself with questions about sound cards, or even if sound devices will be present when the user runs the application.

Hardware advances, in the form of better sound cards with digital signal processing (DSP) ability, are just over the horizon. To take full advantage of the upcoming hardware advances, we need better methods of describing the aural environment.

Some prior work in sound for virtual environments concentrated on spatialization of sound. Other work concentrated on psychoacoustics and the acoustical environment. The objective of this research is to develop a method of representing audio, both spatialized sound and the acoustical environment, in such a manner that it integrates seamlessly into the graphical environment, in both the modeling and run-time domains. Specifically, this research bridges the gap by addressing the following points:

- Traditional model building is the creation of a visual model. If audio is to exist, it must also be created. Why are these two not built together?
- The virtual environment developer generally does not built for a specific piece of graphics hardware; instead, the program addresses a high-level application interface that knows how to address various graphical hardware. If a particular machine cannot support texturing, then the environment will be displayed without textures. In a similar vein, the virtual environment developer should not need to build for any specific sound hardware.

C. SCOPE

The scope of this thesis is the development of a method of representing the acoustical properties of a virtual environment. The representation must fit seamlessly with the graphical development. This thesis will thoroughly describe a process for acoustic modeling and rendering that is seamlessly integrated into the geometric modeling process. A proof of concept implementation is also presented but is not to be considered a production system.

The proof of concept for this thesis will consist of the following:

- A rudimentary graphical user interface (GUI) based modeling tool
- A rudimentary audio rendering tool (demo)

This thesis does not address the following:

- Real-time computation of inherent sounds (i.e. scratching, dropping, etc)
- Sonification
- Sound rendering
- A production version of software

D. ASSUMPTIONS

This thesis delves into the depths of acoustics and psychoacoustics. The thesis also addresses graphical environments and virtual worlds. Although the gist of these subjects is covered in the section II BACKGROUND, this research is better understood with a greater understanding than a cursory overview can provide. If the background section does not provide enough information for the reader to grasp the fundamental concepts on which this thesis is based, Rossing's *Science of Sound* [ROSS90] covers sound in a most thorough manner. Also, for a better understanding of the graphical portion of virtual environments, the book *Computer Graphics Principles and Practice* [FOLE97] is premiere.

In preparation of this research, a thorough literature review has been performed. The results of this review have been instrumental in preparing this research. An annotated list of references is contained in the Bibliography.

E. THESIS ORGANIZATION

This thesis is divided into seven Chapters. Chapter I introduces the topic with motivation and overview. Chapter II provides background into audio and its principles. Chapter III explores real time sound and other works related to this thesis. Chapter IV discusses the approach used in the development of the proof of concept. Chapter V describes the implementation of the proof of concept. Chapter VI contains discussions about why things were developed as they were and how well the proof-of-concept satisfies the stated requirements. Chapter VII reveals the thesis conclusions, recommendations, and future work. The thesis is concluded with the List of References and Bibliography.

II. BACKGROUND

A. FUNDAMENTALS OF SOUND

When you throw a rock into the water, the water at the point of impact becomes disturbed. Upon impact, the water will be pushed out of its place. Once the rock travels down, the water will rush in to fill the void. The water will begin to move in a rhythmic motion. From the point of impact, waves will develop, traveling away from the point of impact in a rhythmic fashion. If the rock is tossed lightly, the waves have a small amplitude; if it is thrown with more force, the waves will have greater amplitude. If the rock is small, the waves will have a high frequency and be close together, while a large rock will cause a lower frequency wave that is more spread out. Now, let's compare this to a sound wave.

If you were to throw a book onto the floor, causing it to land flatly, the air at the point of impact will become disturbed. The waves that are generated by the rhythmic motion of the air are called sound waves. When sound waves reach the ear, they are interpreted as sound. A sound wave is shown in Figure 1.

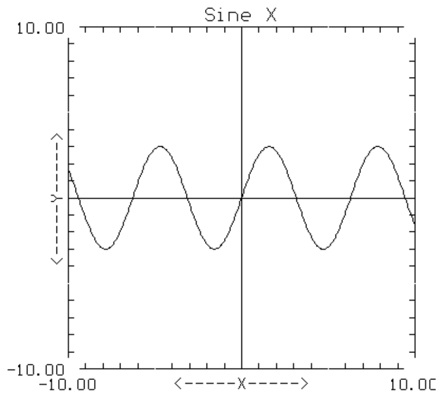


Figure 1: A Sound Wave.
The pitch of this sound wave is represented by one complete cycle of the wave.
The loudness is represented in the amplitude of the wave.

When a sound wave reaches the ear, it has three main characteristics: frequency, intensity, and harmonic structure. When sound waves are interpreted, their three main characteristics are pitch, loudness, and timbre. The three characteristics of sound and sound waves are so closely interrelated that we must address them together even when describing them separately.

1. Pitch

From an egocentric viewpoint, pitch is the auditory characteristic that people use to order sounds on a musical scale. [WELC98] The pitch of a note is logarithmically proportional to the frequency of a note. As the frequency of the sound rises, the pitch is perceived to be higher, as depicted in Figure 2. A sound's pitch can also be affected by the sound's other characteristics. The loudness of the sound can affect the perceived pitch; louder notes are generally perceived as higher notes, within a given range. The complexity of the sound, determined by its harmonic structure, can also affect the way the pitch is perceived.

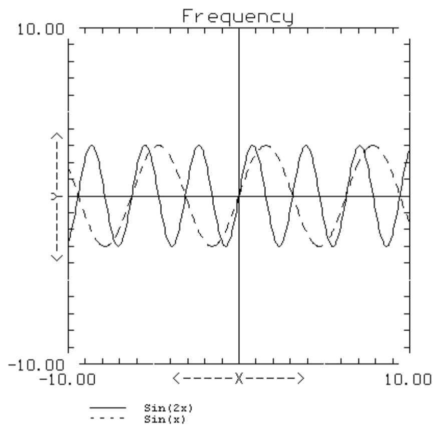


Figure 2: Frequency
The dotted wave is the same wave from Figure 1. The wave represented by the solid line has a frequency twice that of the first wave, represented by the dotted line. Musically, the second note would be pitched one octave higher than the first note.

2. Loudness

Loudness is the attribute of sound that enables us to order the sounds on a scale of soft to loud. In a graph, loudness is represented by the wave's amplitude. A loud note's wave has a high amplitude. A soft note's wave has a small amplitude. Pitch, however, can also affect the perceived loudness of the sound. Of notes in a given range, the higher note is generally perceived as louder. When loudness is being considered, the more complex sounds in a given range are usually perceived as louder sounds.

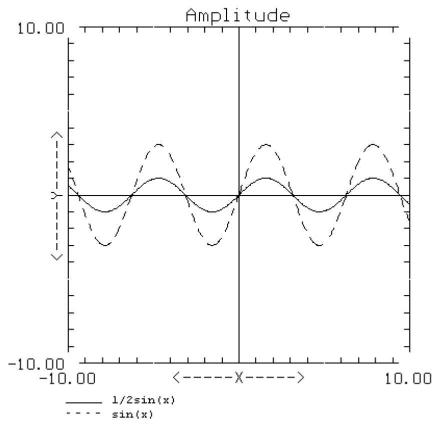


Figure 3: Amplitude
The dotted wave is the same wave from Figure 1. The wave represented by the solid line is only half as loud as the first wave. The second wave would sound much softer than the first wave.

3. Timbre

Timbre is the quality that allows us to distinguish one sound from another, and to group sounds into related families or groupings. Timbre is sometimes referred to as everything about a sound that is not pitch or loudness. [BEGA94] [ROSS90] Although we do not have a complete model for fully specifying timbre, we have defined two of its most important characteristics, *sharpness* and *compactness*. The sharpness of the sound relates to the energy concentration of frequency spectrum. Sounds with much energy in the high end of the spectrum sound sharp, while sounds with more energy in the lower end of the spectrum tend to sound muddy. The compactness of a sound is determined by the distinction between the discrete harmonics of the tone as compared to the continuous harmonics of noise.

Since timbre is not a well-defined attribute, an in-depth discussion of its characteristics is appropriate.

A complex tone is a tone composed of more than one sine wave. The lowest wave of the tone is called the fundamental frequency, or fundamental. The fundamental is the pitch that we recognize when we hear the tone. The first partial, or first harmonic, is the sine wave whose frequency is the same as the frequency of the complex tone.

The complex tone is also comprised of other sine waves called partials. The second partial, or second harmonic, in a complex tone is twice the fundamental, or one octave above the fundamental. The third partial is three times the fundamental, the fourth partial is four times the fundamental, etc. The addition of the harmonics is what gives a complex tone its unique sound.

By adding different harmonics, or changing the loudness of the harmonics, we can alter the timbre of the sound. We can group the harmonics to get better control of the timbre. The second harmonic is almost inaudible, but makes the sound fuller. The third harmonic produces a *blanketed* sound, making the sound softer.

The lower harmonics, first through sixth, group themselves into even harmonics (second, fourth, and sixth) and odd harmonics (third and fifth). The odd harmonics produce a *stopped* or *covered* sound. The even harmonics produce a *choral* or *singing* sound. A strong second and third will *uncover* the sound. A strong third and a strong fifth give the sound an annoying, metallic color. A strong second, third, fourth, and fifth produce a french-horn type sound.

The high harmonics, seventh and above, give the tone its edge. If the upper harmonics do not overpower the fundamental, they reinforce it, giving it a strong *attack*. Overpowering upper harmonics can make the sound raspy. [HAMM98]

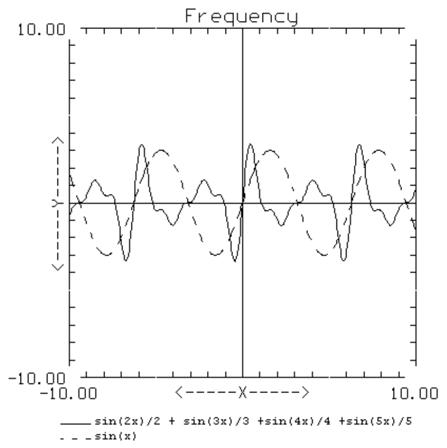


Figure 4: Timbre
The dotted wave is the same wave from Figure 1. The wave represented by the solid line has a much more interesting timbre. It is composed of the original wave plus the next four partials.

B. PSYCHOACOUSTICS

1. How do we hear sound differences?

An important facet of our perception of sound is the relative difference between sounds. If a sound is played, then another sound is played, how close in pitch do those two sounds need to be for the average person to recognize their pitches as different? This characteristic is the *just noticeable difference* (JND). The JND for pitch varies depending on where the pitch is located on the spectrum. At low frequencies, changes of two to five Hz are detectable when pitches are played consecutively. When pitches are played simultaneously, the JND can range from 1-3 hertz (Hz) in the low frequencies. Gelfand showed that the pitch JND is directly proportional to the frequency of the pitch. [WELC98] [GELF81] The JND for loudness varies according to the loudness of the original sound. For very soft sounds, a small increase in volume is recognizable. As the initial loudness increases, the JND increases.

Masking is another factor in our perception of sound. Masking is the amount that a threshold of audibility is raised by the presence of another sound. An example is road noise masks the sound of a car stereo. When you pull off the interstate and come to a stop, your stereo seems much louder. Masking tends to occur between sounds that are close in frequency. Lower frequencies mask higher frequencies. Masking also occurs when the audio signals from the two ears converge. For this reason, spatially locating two voices into two distinct, separate locations usually makes both conversations intelligible. [WOUD97]

2. How does the ear hear the pitch?

The basilar membrane is the part of the ear inside the cochlea that responds to and converts sound waves into signals that the brain can interpret. The basilar membrane contains approximately 10,000 small hairs that respond to sound waves, generating a signal that the brain will receive. How that signal is generated is not known in fact, but two theories have been advanced that attempt to answer this question.

Place Theory, developed by Georg Von Bekesy, states that different points on the basilar membrane undergo maximum displacement as a function of the frequency of the sound wave entering the ear. Each neuron, attached to a small hair, would represent a particular pure frequency. In the case of complex tones, numerous neurons would be stimulated, giving the signal complexity. For frequencies below 400Hz, however, the entire basilar membrane is stimulated equally. This would seem to contradict the average human's ability to hear down to approximately 20 Hz. [WELC98B]

Temporal Theory hypothesizes that the basilar membrane moves up and down with each sound wave. Each up or down motion causes the neuron to fire. The pattern of neurons firing determines the pitch. The neurons in this area cannot fire more than about 1000 times per second. With this theory, our hearing would be limited to approximately 1000 Hz, but the average human can hear frequencies up to 14,000 - 20,000 Hz. [ALBE97]

An anomaly that neither of these theories addresses is that of the missing fundamental. When a sound enters the ear, the complete sound, to include partials, is processed. Schouten discovered in 1940 that people could make a correct pitch judgment when the fundamental frequency was not present, if some of the upper partials were present. The auditory system seemed to know what fundamental should have been present with the partial. For this reason, when an environment contains masking noises, a complex tone is easier to correctly interpret than a pure tone. [ALBE97]

3. How do we organize sounds?

When many sounds surround us, we do not generally hear them as distinct, unrelated phenomenon; instead, we tend to group sounds using both perceptual and physical factors.

Similarity, or lack thereof, is a perceptual factor used to group sounds. Sounds will often be grouped as a single source when they are similar in pitch, timbre, loudness, or location. Sounds which occur close in time, i.e. two short beeps, will be assimilated into groups. Sounds which occur in the same frequency range, like two whistle blasts, are often grouped together. Dissimilar frequencies, like a shrill whistle and a tuba, tend to not be grouped together. Whenever sounds change in pitch or loudness, if this change is smooth it will usually be interpreted as the same sound source. If the pitch or loudness change is abrupt, the change will often be interpreted as a new sound source. Sounds that are grouped together also tend to change in coherent ways. When the pitch changes, the loudness should change in a consistent manner.

The physical factors of the sound are also important for grouping sounds. The fundamental frequency of a sound is very important. If sounds of different fundamental frequencies are played, the harmonics of the individual sounds will not be confused. If the two sounds have the same fundamental frequency, the harmonics will merge, being perceived as a single sound with a timbre different than either of the original sounds.

Sound location is also important for grouping sounds. Sounds that originate from similar locations in space tend to be perceived as sources. Sounds emanating from dissimilar locations tend to be perceived as separate. The sound location factor contributes to the *Cocktail Party Effect* [ARON92][WELC98], whereby a person can listen to two conversations if they are located in different positions in space, but the same conversations tend to be less comprehensible when they occur in the same place in space.

Rhythmic patterns tend to be perceived as a source. According to Deutsch, rhythm is one of the most powerful physical factors of pattern recognition. [DEUT80] [WELC98A] When using complex tones, rhythmic gaps of only two milliseconds are recognizable.

Rhythms can be developed using many methods. [FRAI82] [WELC98] Intensity accentuation is using an accented sound to begin a group. If every third sound has more intensity than the other two, then those sounds will be perceived as a group of three sounds. When playing similar sounds, adding a long interval after every fourth sound will cause the sounds to be perceived as groups of four. Groups of two, three, or four seem to be most natural, while groups of five or more tend to be harder to recognize.

Listeners also use cultural knowledge of scale and key structures of music to group sounds. Dewar, Cuddy, and Mewhort found that subjects could differentiate between two patterns of tones if the first pattern all belonged to one scale and the second pattern all belonged to one different scale. [DEWA77] [WELC98A]

C. PRIMARY LOCALIZATION CUES

Once a sound is created, it is affected by the world around it. The air affects the sound’s ability to travel. Obstacles cause portions or the entire wave to be reflected. Our ears can detect all of these changes to the sound, and the changes have specific meanings when interpreted by the brain. We must discuss this phenomenon to round out our introduction of sound.

Sound travels at approximately 1100 feet per second. If a sound occurs ten feet away, we will hear that sound in 1/110 of a second. If the difference between our ears is one foot (to make the mathematics easier), let us see how long it takes for the sound to reach each ear. As shown in Figure 5, the sound source occurs 10 feet from the left ear, and it occurs 10.73 feet from the right ear. The left ear will hear the sound in .00909 seconds, and the right ear will hear it in .00975 seconds. Our brain uses this information to tell us where, in a left/right area, a sound occurred. This is called the Interaural Time Difference (ITD). [AURE98]

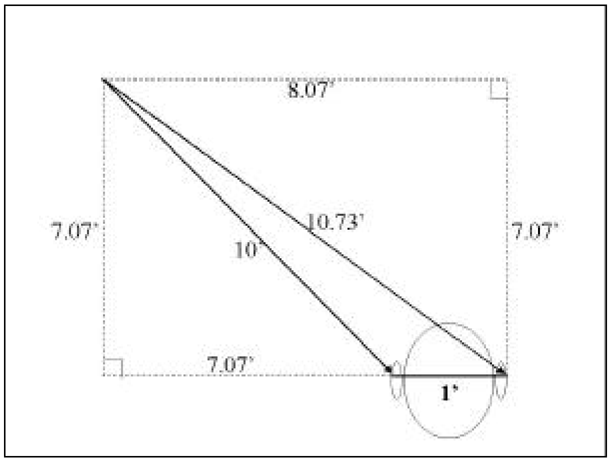


Figure 5: Interaural Time Delay (ITD)

Another effect that we use to locate a sound in the left/right plane is called the Interaural Intensity Difference (IID). As we can see by Figure 6, the head serves to shadow the sound reaching the right ear. As a result, the intensity at the left ear is greater than the intensity at the right ear. By combining the ITD and IID cues, we are able to narrow the position of a sound. [AURE98]

The third effect used to locate the sound is actually caused by the listener's own ear. The pinna, or the outer ear, filters the sound as it enters the ear canal. As a sound moves in elevation, the spectral filtering of the pinnae changes. An identical sound coming from two different elevations will have different timbres. In effect, the pinnae act as variable filters. As we learn to hear, we learn how our pinnae affect the sounds we are hearing. Most listeners have learned to map the pinnae's spectral filtering to the sound's correct elevation.

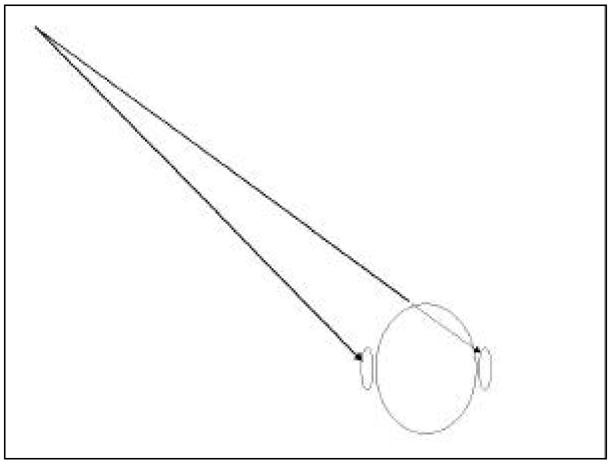


Figure 6: Interaural Intensity Difference

The combination of ITD, IID, and the pinnae's spectral filtering allow us to precisely locate a sound in three-dimensional space. [AURE98]

D. SOUND DIFFUSION

The final characteristic that we must cover is the diffusion of sound in a room. To say that a sound is perfectly diffused in a room is to say that the sound's pressure is the same at all points in the room at the same time. This is neither achievable nor desirable; however, without some diffusion, we would not hear sounds throughout the room.

The terms *live* and *dead* are terms often used to describe the amount of diffusion in a room. A qualitative term, *liveness* depends on the ratio of reflected sound to direct sound. The higher the ratio, the more live the room is. As one moves away from a sound source, one perceives the liveness to rise; conversely, as one moves closer to a sound source, one perceives the sound to be more *dead*.

Diffusion of a sound in a room can be affected by modifying the ways that sound can be scattered. Irregularities in walls cause reflecting sounds to reflect in a more scattered pattern than a smooth, polished wall. Chairs, lamps, and other objects, to include people, scatter sound that comes into contact with them. Placing more objects that scatter sounds will increase the liveliness of the room. [KNUD50]

Acoustically absorptive material, on the other hand, will lessen the amount of reflected sound. When sound strikes acoustically absorptive material, most of the sound is reflected. The unreflected sound, however, is absorbed and transferred into heat. Adding a material such as celotex tile to a room will make the room more *dead*. These concepts are shown in Figure 7.

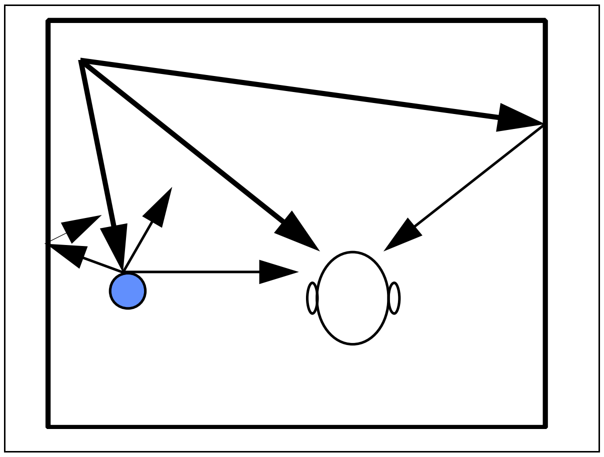


Figure 7: Sound Diffusion
The sound is emanating from the top left corner of the room. As the sound strikes a surface, it is partially absorbed and partially reflected.

E. VIRTUAL SOUND

Real sounds occur in the real world when real objects cause other real objects to vibrate. Virtual sounds, on the other hand, are sounds that are created by synthetic methods. An electronic keyboard could be considered creating virtual sounds. Playing a Compact Disc (CD) or cassette tape would be a virtual sound, even though it is a recreation of a real sound. For the purposes of virtual environments, we want to concentrate on virtual sounds.

One of the advantageous aspects of virtual sounds, whether they are generated (synthetic sounds) or recorded, is that they can be processed using standard audio post-production techniques to give a three-dimensional quality.

F. AUDIO ENGINEERING

Using standard audio processing equipment, such as an echo or delay unit, echoes and delays can be added to a sound. Using the distance and angle from the object to each ear of the listener, we can calculate the time delay required for the sound to reach each ear. Using the same distance and angle, we can calculate the appropriate sound pressure level for each ear. If we are extremely industrious, we can also calculate the effect the pinnae will produce. [See section II. C.] Using this information, we can use audio delay units, like the Rane AD 22 Audio Delay [RANE98A], and equalization units like the Rane GE 60 Stereo Interpolating Constant-Q Equalizer [RANE98B], to alter the original virtual sound, giving us a sound that appears to originate from the object's location.

When performed in a modern post-production facility, like a recording studio, the above mentioned scenario is not as difficult as it sounds. Even so, it is not a process that happens in real-time. To accomplish something like this in real-time, we need to turn to Digital Signal Processing (DSP) hardware.

G. DSP HARDWARE

The audio processor provides many features to support a variety of audio applications. The following is an overview of the features a few popular audio processors found on the market today:

1. Audio Processor Features of the SGI Indigo

a. General Features

- Independent input and output sample rates
- Simultaneous input and output of audio data to/from applications
- Multiple applications sending and receiving audio data
- Input audio from one of microphone, line in, or digital input
- Output simultaneously to all of headphone/loudspeaker, line out, and digital out

b. Digital Signal Processor

The original Indigo contains a Motorola 56001 Digital Signal Processor (DSP). This processor is used to maintain real-time flow of audio data to and from the

MIPS R4000 processor, and to perform mixing operations between audio applications. Indigo2 and Indy do not contain a dedicated DSP chip; instead, the operation of the audio system is split between the HAL2 ASIC, HPC3 ASIC, and software running on the MIPS CPU. [IRIS96]

c. Audio CPU Usage

As noted above, the Indigo2 and Indy no longer contain a dedicated DSP chip. Thus, the impact of some Audio Library functions upon system performance has changed. In particular, each audio port that remains open consumes a small but relatively constant amount of the MIPS CPU. Output ports tend to be more expensive than input ports, since they require the kernel to perform mixing on behalf of the application. [IRIS96]

2. Creative Labs

Creative Labs Sound Blaster Live!™ is Creative Labs' newest audio card. The Sound Blaster Live! produces sound for up to four speakers, and when used with the Environmental Audio Extensions (EAX) API, can route sound to Dolby 5.1 or MPEG-2 7.1 speaker configurations. This sound card contains the EMU10K1™ audio processor chip. It provides real-time for three-dimensional sound placement, and many real-time digital effects like reverb and pitch shifting. Live! accelerates Microsoft DirectSound® and DirectSound3D®, and contains user-selectable DSP modes for acoustical environments such as Hall, Theater, Club, etc. [SBLI98A] Live! boasts an average noise floor level of –120dB. [SBLI98B]

a. General Features

- Supports real-time digital effects like reverb, chorus, flange, pitch shifter or distortion across any audio source
- Capable of processing, mixing and positioning audio streams using up to 131 available hardware channels
- Customizable effects architecture allows audio effects and channel control
- Full digital mixer maintains all sound mixing in the digital domain, eliminating noise from the signal
- Full bass, treble, and effects controls available for all audio sources
- 64-voice polyphony with E-mu's patented 8-point interpolation technology
- 192-voice polyphony PCI wave-table synthesis
- 48 MIDI channels with 128 GM & GS-compatible instruments and 10 drum kits

- Uses SoundFont® technology for user-definable wave-table sample sets; includes 2MB, 4MB and 8MB sets
- Load up to 32MB of samples into host memory for professional music reproduction

b. Digital Signal Processor

- E-mu Systems EMU10K1 patented effects processor

3. Aureal family of sound cards

The Aureal family of sound cards specializes in three-dimensional sound. The newest member of the family, the Vortex II, offers silicon based processing of audio data. It is capable of driving headphones, or speaker configurations of two to eight speakers. The Vortex II offers hardware based sound acceleration for A3D, Aureal's proprietary applications programmer's interface (API), and for DirectSound3D, Microsoft's open API. [AURE98B]

a. General Features

- Hardware-based A3D and DirectSound3D engine (16 sources)
- Hardware-based A3D 2.0 Wavetracing engine with wall reflections and occlusions (64 sources)
- 320-voice DLS wavetable engine for DirectMusic and MIDI (64 hardware, up to 256 software voices depending on CPU)
- Hardware DirectSound engine (96 sources)
- Optimized for headphone, 2-speaker, or multi-speaker (up to 8) playback
- Hardware-based 10-band stereo graphics equalizer (96 dB Signal to Noise Ratio)
- Legacy audio support in real-time DOS and DOS boxes
- Hardware-based crosstalk cancellation
- 96 DMA channels
- Windows 98/95 and NT 4.0 Drivers (WDM ready)
- Full MIDI I/O and DirectX gameport acceleration
- SP/DIF output
- Aureal Wavetracing™ Technology: real-time acoustic reflections, reverb, and occlusion rendering.

b. Digital Signal Processor

- Vortex 2 (Part No. AU8830)

4. Acoustetron

The Acoustetron II is a stand-alone 3D sound server system from Crystal River Engineering, Inc., a subsidiary of Aureal. The Acoustetron II is a close predecessor of the Vortex II and A3D sound. It consists of a 486DX4 PC host computer with four DSP cards containing a Motorola DSP56001. The system performs real-time spatialization of multiple real-time audio sources in three-dimensional space. When appropriately configured, it includes the ability to render four concurrent 3D sound sources with six reflections each at 44kHz sampling rate. Furthermore, any or all of the six reflective surfaces can be *textured* by applying a material to the surface. The surface can also be modified with an amplification value to further refine the acoustical environment. [ACOU96]

	Crystal River Engineering	**Aureal**	**Creative**	**SGI**
	Acoustetron II	Vortex II	SoundBlaster Live	Indigo
Max Sampling Rate	44.1kHz	48kHz	44kHz	48kHz
S/N		>90dB	100dB	
DSP	Motorola DSP56001		E-mu Systems EMU10K1	
MIDI	Yes	Yes	Yes	Yes
3D Audio Capable	Yes	Yes	Yes	Yes
Audio Texturing	Yes	Yes	No	No
Speaker Output	Headphone, 2 Speaker	Headphones, 2-8 speakers	Headphones, 2/4 speaker Mode, Dolby 5.1 capable MPEG-2 7.1 Capable	2/4/8 Speaker output
Wall Reflection	Yes	Yes	No	No
Occlusion	Yes	Yes	No	No
Effects	Pitch shift	Pitch Shift, Reverb	Reverb, chorus, flange, pitch shift, distortion	
Other			10 band graphic EQ in hardware	
APIs supported	CRE_TRON	A3d, DirectSound, DirecSound3D	DirectSound, DirecSound3D, Environmental Audio	AL

Table 1: Comparison of Sound Card Features

H. WHAT IS IMPORTANT?

Given the above, we now have a multiple-entrypoint question to address. The question is, "What is important?". The question can be addressed in the following ways:

- If I know the human performance requirements of my application, what do I do?

- If I have a hardware/budgeting constraint, what is the best performance I can achieve?

The following table represents the above mentioned question:

	Performance requirement	**Hardware requirement**
1	No sound	No additional gear
2	System warning prompts	Standard audio speaker supplied with a current computer
3	Non-directional audio, to include sounds, speech, warnings	Sound card Speakers or headphones
4	Directional audio for single user, no headtracking	Sound hardware with 3D capability Headphones or speakers
5	Directional audio for single user, with headtracking	Headtracking device Sound hardware with 3D capability and fast, interactive positional update
6	Directional audio for multiple users	Multiple 3D capable devices with headphones
7	Directional audio for multiple users, with headtracking	Multiple 3D capable devices with headphones and headtracking devices

Table 2: Technology Requirements

III. REAL TIME SOUND/PREVIOUS WORK

Now that the basics of sound have been discussed, let us delve into how sound is represented in the virtual environment. First, we will look at previous work in three-dimensional sound at NPS. Second, we will look at how sound is created in a three-dimensional virtual environment. Third, we will look at some Application Programmer's Interfaces (API) that allow a three-dimensional virtual environment to address the sound generation hardware. Finally, we will look at some current families of modeling and rendering software.

A. RELATED RESEARCH

1. Implementing 3D sound into specific virtual environments

U.S. Army Major Russell Storms implemented a MIDI based sound server for three-dimensional sound in NPSNET. This implementation took advantage of the cost efficient sound generation MIDI hardware, and brought three-dimensional sound to NPSNET at a very low cost. This thesis did not address the use of an aural environment or aural geometry. [STOR95]

Integrating Realtime 3D Sound Into NPSnet, by Marine Corps Captain Lloyd Biggs, addressed the integration of three-dimensional sound produced from *wav* files stored on a sound server. This thesis addressed a specific implementation of 3D sound in an effective manner. It did not, however, address a method of building an aural environment in which 3D sounds could exist. [BIGG96]

Hesham Fouad developed a Virtual Audio Server (VAS) for the dispatching and scheduling of sounds. VAS implements techniques for managing overload conditions in a sonic virtual environment. Real-time scheduling algorithms allow for graceful degradation of sounds according to the algorithms set by the programmer. [FOUA97] In the graphics world, VAS would correspond to the CULL section of the APP-CULL-DRAW process. As such, it belongs in the audio rendering pipeline. The APP-CULL-DRAW process is discussed in depth in the Approach, beginning on page 33.

The Helsinki University of Technology's *Acoustics Laboratory* and the *Telecommunication Software and Multimedia Laboratory* have worked on the Digital Interactive Virtual Acoustics (DIVA) project for many years. The aim of DIVA is to create both visual and aural illusion in virtual three-dimensional space where a person can move around freely and interact with his/her environment. The main goal of DIVA is to study real-time virtual audio in its many different forms, to include computational modeling of room acoustics, physical modeling of musical instruments, spatialization and auralization, and many others.

DIVA have produced outstanding applications such as the Virtual Orchestra, where a human can, in real time, conduct an orchestra composed of computers that physically model instruments in real time. Most recently, DIVA produced Marienkirche, a visual and aural demonstration film. Marienkirche is a film rendered with **3D StudioMax** and **Lightscape** software for the visual, and **DIVA** software for the aural. The film accurately depicts walking through the cathedral of St. Marienkirche, a 13th century gothic cathedral destroyed at the end of World War II. [DIVA98] It is an outstanding example of what all virtual environments should strive to imitate.

The DIVA project has produced outstanding work in 3D audio, but it has not effectively tied the development of the audio environment to the development of the visual environment.

2. Real-time sound generation

Timbre trees were introduced as a method of generating sound. Timbre trees represent a sound as a tree composed of the mathematical functions that would create the sound. By instantiating a timbre tree with the proper parameter variations, different sounds can be created. Adjusting data values in a timbre tree can modify the sound produced by the tree. Connecting multiple timbre trees, as in a collision of two objects, could possibly create appropriate sounds for the collision of the two objects. [FOUA97]

Michael Casey, Machine Listening Group, Massachusetts Institute of Technology, developed Perceptual Audio Models (PAM), a program that encodes *classes* of sounds. Instead of a standard sound sample, PAM determines the most important characteristics of a sound, then saves that information. By combining the characteristics of a rubber hammer with the characteristics of a bell, a user can synthesize the sound of a rubber hammer striking a bell. [BEAC97]

The Musical Instrument Digital Interface (MIDI) is a protocol for communicating with musical instruments, to include sound synthesizers. MIDI is a low bandwidth method of creating sounds on any standard PC sound card.

Sampling is the method of recording sounds to digital disk. Many people are already accustomed to working with *wav* files on the PC. Once a sound is sampled, the sound can be manipulated or played back. Sample playback, or playing a wav file, is another method of creating sounds for a virtual environment.

Timbre trees, PAM, MIDI, and *wav* files are all methods for sound generation. In an audio-equivalent of the APP-CULL-DRAW process, sound generation belongs in the APP section. Although this is a very important topic, it is not within the scope of this thesis.

B. DSP APPLICATION PROGRAMMER'S INTERFACES

Many flavors of sound producing hardware exist for both the PC and SGI. To address the hardware, manufacturers supply software developers with a software development kit (SDK). The SDK includes an Application Programmer's Interface (API), which includes the data types and function calls that address the hardware.

Many APIs exist, but only a few are common. The next section discusses the major APIs that exist for sound hardware.

1. DirectSound

The DirectSound and DirectSound3D APIs are Microsoft's APIs for sound in the DirectX system. DirectSound is the API for non-directional sound, while DirectSound3D is the API for three-dimensional sound. They require only a standard Microsoft Windows compatible sound card. DirectSound addresses the audio in the same manner that DirectX addresses the visual geometry. Sounds can be attached to visual geometry, and when the geometry is moved the sound automatically moves with it. This is the preferred method. DirectSound addresses the acoustical environment by predefining a few acoustical environments. DirectSound has definitions for rooms and halls.

DirectSound did not, however, include specifications for acoustical absorption properties. These absorption properties could be attached to a piece of geometry when the geometry is textured. The room acoustics could then be calculated by adding the absorption properties of the different objects in the room. This is the only part that Microsoft left out. DirectSound is supported by all Microsoft Windows compatible sound cards.

2. A3D

A3D is a three-dimensional audio API from Aureal Semiconductor. A3D allows for three-dimensional placement of sounds at any distance and position from the listener. When run with an A3D compatible sound card (a requirement) it adds less than 5% CPU usage on an Intel Pentium 166 with eight sounds playing. [AURE98C] A3D accomplishes Distance modeling with atmospheric filtering and gain. All positioning (left/right, up/down, front/back) is accomplished using head related transfer functions (HRTF). Occlusions are facilitated with gain, and when coupled with A3D 2.0 drivers, material filtering is also applied. Reflections are not addressed in A3D. [AURE98C]

3. A2D

A2D is a speed optimized, feature reduced version of A3D, designed to emulate A3D in software. A2D allows A3D 2.0 applications to run on PC platforms that do not have A3D hardware support. It uses gain for distance modeling and Front/Back modeling. It uses ITD and IID for left/right positioning. When coupled with A3D 2.0 drivers, it uses gain to model occlusions. [AURE98C]

4. A3D 2.0

A3D 2.0 is Aureal Semiconductor's most modern three-dimensional API. It accomplishes distance modeling with atmospheric filtering and gain. All positioning (left/right, up/down, front/back) is accomplished using HRTF. Occlusions are facilitated with gain and material filtering. Reflections are modeled using HRTF, reverb, and material filters. A3D 2.0 also includes intelligent resource management, sound source priorities, audio culling, room geometry culling, and reflection management. [AURE98C]

5. Environmental Audio/EAX

EAX is Creative Lab's extension to the original DirectSound API. Creative Lab's position is that programmers should not be locked into a proprietary API. Instead, the sound cards should respond to a non-proprietary API. [SBLI98C] Creative Labs is betting on Microsoft's DirectX API.

6. SGI Audio Library

The SGI Audio Library (AL) is a low-level API for addressing a specific set of hardware, specifically the SGI audio hardware. It was designed to enable multiple programs to share the audio resources of the workstation, and programs written on any SGI machine with audio can correctly execute on any SGI machine with audio capability. [IRIS96]

The Audio Library provides three major capabilities:

- input and output of digital audio data
- control of the attributes of the digital audio data
- control of physical parameters of the audio subsystem [IRIS96]

	DirectSound3D	**A3D 2.0**	**EAX**	**AL**
Distance Modeling	✓	✓	✓	
Left/Right Positioning	✓	✓	✓	
Front/Back Positioning	✓	✓	✓	
Elevation Positioning	✓	✓	✓	
Occlusion		✓		
Reflection		✓		

Table 3: Comparison of APIs

Although the API in itself is not the focus of this thesis, a look at the APIs gives us a feeling of what can be accomplished and how it can be done. As Table 3 illustrates, the A3D API addresses the acoustical environment very strongly. The DirectSound3D and DirectX API address the incorporation of sound into the graphical environment better than the other APIs. An overarching API should be developed so that a programmer needs to write only one set of code. If the calls are not supported by the hardware/API installed on the machine, then those calls are either interpreted or dropped.

C. FAMILIES OF MODELING AND RENDERING TOOLS

Many commercial companies develop applications for modeling and for the rendering of the models. Coryphaeus Software, Inc, and Paradigm Simulation, Inc, are two companies that develop families of tools for modeling and rendering processes. By developing families of tools, the transition between the modeling phase and the rendering phase is simple because of the integration of the products.

1. Coryphaeus Software, Inc.

Coryphaeus Software, Inc., develops modeling and rendering tools for the SGI. Designer's Workbench™ (DWB) is an interactive database modeler with GUI. It allows a user to develop geometry, add texture, add materials, colors, and sound. It addresses sound down to the Audication model. [The Audication model is proposed by this thesis, see page 38.]

EasyScene™ is a graphical rendering program designed to load model databases built with DWB or many other modeling tools. It allows the user to walk through the database, performs collision detection, renders fog, 3D spatialized sound, and most other options found in a graphical rendering program. As with DWB, EasyScene addresses sound down to the Audication model, but does not address the Aureflexion model. [The Aureflexion model is proposed by this thesis, see page 38.]

2. Paradigm Simulation, Inc.

Paradigm Simulation Inc. has built a complete family of modeling and rendering tools that allow rapid prototyping, building, editing, and rendering of sophisticated environments and applications quickly and easily.

Vega™, the simulation development tool for geometry, allows for the development of advanced three-dimensional geometry, to include lighting models and special effects. It is based on SGI's Performer™, so it is an extendable tool. If more power is needed, programmers can access a C language application programmers interface (API) to access Performer, OpenGL™, and Paradigm's graphics libraries.

AudioWorks2™ is Paradigm Simulations' audio environment development software. AudioWorks2 supports full three-dimensional audio rendering using Crystal River Engineering's (CRE) Acoustetron II. AudioWorks2 allows a user to develop an acoustical environment, to include both the Aureflection and Audication model, and to hear that environment in real time. Using AWLynX, the sounds can be connected to sounds in the graphical world, so that as transforms are applied to geometry, the same transforms are applied to the sounds attached to the geometry.

Both Vega™ and AudioWorks2™ use the LynX graphical user interface. LynX provides the ability to attach audio objects to visual objects and to duplicate most of the functions found in the API. Using Vega-Audio, some classes in Vega and AudioWorks2 can share objects, allowing a single object to be created with both visual and aural attributes.

Paradigm Simulations has correctly addressed both the Audication and Aureflection models; however, they have chosen to separate the visual world from the audio world for development purposes. This allows each developer to concentrate on one side of the puzzle without concern for the other. This works well for programming teams having at least one member with a strong background in acoustics. For programmers without extensive expertise in audio, this adds a level of difficulty that is unnecessary.

3. Virtual Reality Modeling Language (VRML)

VRML is not actually an API, but it is a very useful, cost-effective method of implementing virtual reality over the Internet. VRML has the ability to bring virtual reality to the common man. Because of the probability of VRML existing in large quantities on the Internet for quite some time, VRML should be considered in this survey.

VRML has chosen to model sound from the perspective of sound generation and sound reception. [Generation and reception are discussed in depth in Chapter IV APPROACH, beginning on page 33.] A sound node can emit a sound. That sound can be spatialized, if the hardware is able. The sound can be somewhat directionalized in elliptical patterns. The sounds can be prioritized, so the web browser knows which sounds to drop if it cannot reproduce everything. Sounds are easily attached to geometry, and when the geometry moves, the sounds move. From the perspective of the sound emission, VRML has done the best job at present.

The acoustical environment [or propogation, see page 33] is not represented in VRML. Although a sound can be spatialized, if the sound is emitted in a small room with walls of wood, the echoes of the sound interacting with the wood are not represented. Furthermore, if the walls of the room were covered with heavy drapes, the sound level in the room would not be affected.

D. QUICK SYNOPSIS AND OPINION

After reading the preceding chapters, the following points should be clear:

- Audio is a necessary, or at least very useful part of a virtual environment.
- Research into three-dimensional audio rendering is advancing it at a rapid pace.
- Three-dimensional audio APIs address sound at a low level.
- Families of modeling/rendering tools work well for visual aspects but not necessarily for aural.

From the research performed, it is apparent that the virtual graphical world has matured more quickly than the virtual audio world. Virtual environments represent the graphics of the environment very well. The graphical world has developed methods for representing imagery (polygons, lines, textures) that work very well for imagery, and these methods have also worked very well for audio, when they have been applied.

It is apparent that when developing a three-dimensional virtual environment, the audio portion of the environment is usually lacking due to difficulties in implementation. It is simply hard to do; consequently, it is often left out altogether. Furthermore, understanding the complexities of a three-dimensional environment is not always possessed by computer programmers. We should develop a method that allows a programmer/modeler to develop the audio properties of the environment seamlessly while developing the graphical environment. It should be integrated, and it should be easy to implement and understand.

In much the same way that Iris Performer and *Open Inventor*™ allow a programmer to write high level code to perform low level OpenGL graphics functions, a method should be developed to allow audio programming to be performed at a high level.

Furthermore, when geometric pieces and audio pieces are connected, as in a pfDCS (Dynamic Coordinate System) in Performer, a single call to move the DCS should move the geometry and the audio. This is discussed in more depth as the Audication model on page 38. Also, a single call to texture a polygon should not only apply the texture to the piece of geometry, but it should also apply acoustical properties as well. This is discussed in more depth as the Aureflexion model on page 38.

IV. APPROACH

A. THE THREE STATE APPROACH

Sound in a real world environment can be considered to exist in three separate states; generation, propagation, or reception, as shown in Figure 8. [FOST98] The generation state exists at the moment the sound is created – it involves sound emission. For an impulse sound, like a hammer striking a bell, this state is instantaneous; for a longer sound, such as an engine running, this state exists for a period of time. Once the sound has been created, it interacts with the environment, bouncing off some objects and being absorbed by others – this is the propagation state where acoustics are considered. Finally, when the sound reaches the listener's ears, the ears receive the sound for further processing as it goes to the brain – this is the reception state, or where the sound is localized in space.

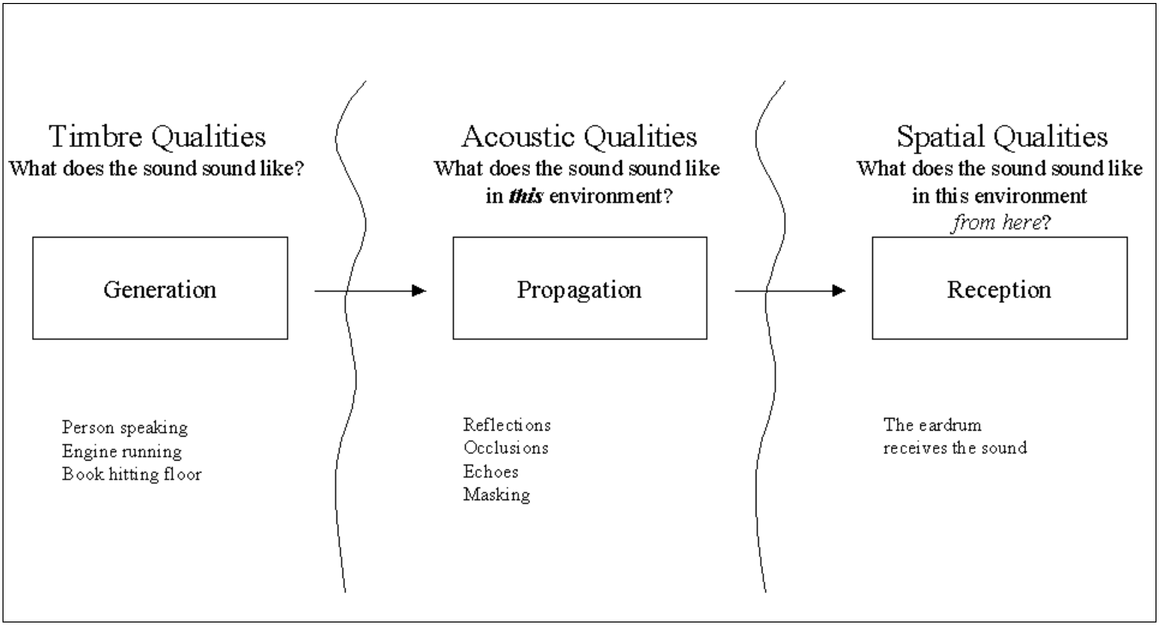
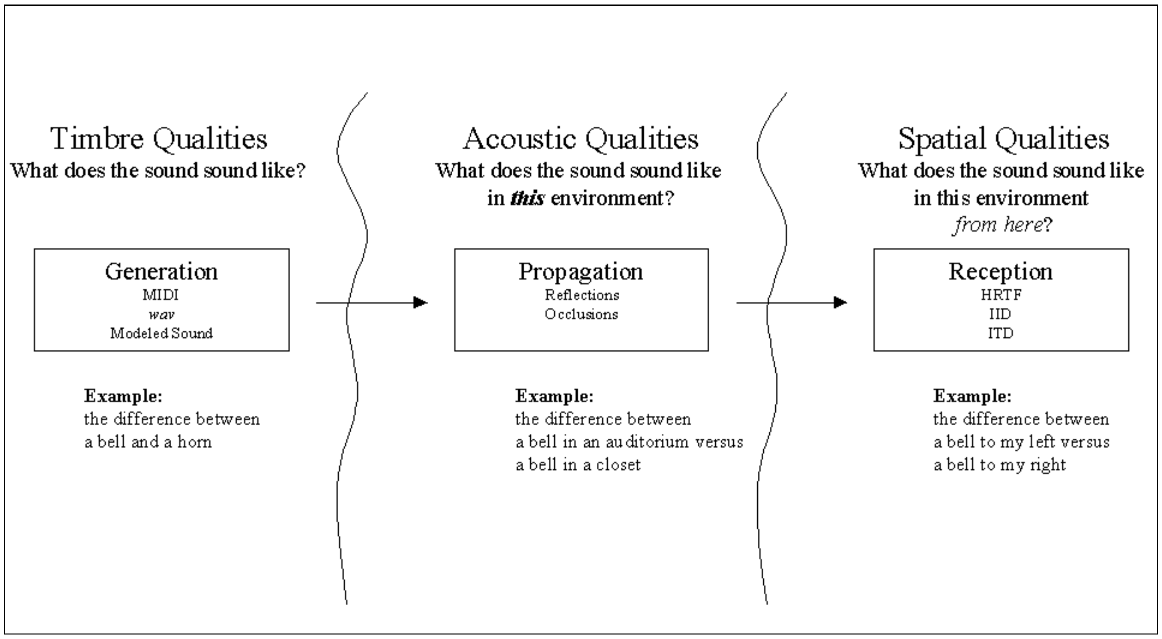

Timbre Qualities
What does the sound sound like?
Acoustic Qualities
What does the sound sound like in *this* environment?
Spatial Qualities
What does the sound sound like in this environment *from here*?
Generation
Propagation
Reception
Person speaking
Engine running
Book hitting floor
Reflections
Occlusions
Echoes
Masking
The eardrum receives the sound

Figure 8: The three states of a sound in the real world.

Much like sound in the real world, sound in a virtual world also can be considered to exist in these three states, as depicted in Figure 9. Initially, the sound must be generated. MIDI and *wav* files are methods of playing back a recorded sound. Timbre trees and sound modeling are methods of generating sounds on the fly in the virtual world. Either of these methods takes place in the application, or APP, portion of a virtual environment. [Refer to section on Virtual Sound, page 16].



Timbre Qualities
What does the sound sound like?
Acoustic Qualities
What does the sound sound like in *this* environment?
Spatial Qualities
What does the sound sound like in this environment *from here*?
Generation
MIDI
wav
Modeled Sound
Propagation
Reflections
Occlusions
Reception
HRTF
IID
ITD
Example:
the difference between a bell and a horn
Example:
the difference between a bell in an auditorium versus a bell in a closet
Example:
the difference between a bell to my left versus a bell to my right

Figure 9: The three states of a sound as represented in a virtual world.

Once the sound is created, it must interact with the virtual environment. If the environment contains any objects, the sound must be reflected from those objects. [Refer to section on Sound Diffusion, page 15] Any large object directly between the sound source and listener would occlude, or lessen, the sound that the listener hears. Effects such as JND and masking [page 11] would also be applied. The intensities of all of the reflections would be calculated, and those that were loud enough would be sent to the sound rendering hardware. The sounds that would not be heard by the listener would be culled out, or not sent to the sound rendering hardware. Graphics programming has a similar construct called the CULL process, which gathers all polygons that will be drawn. If propagation is not considered, the sound exists only in an anechoic environment, which is not a natural state.

Finally, the render, or PLAY, process would take place. In graphics programming, only the polygons that could be seen by the user would be drawn on the screen – this is the DRAW process. In the audio process, only the sounds that can be heard would be played. In addition to playing the sounds, the PLAY process would apply HRTF, or some other method, to spatialize the sounds [as discussed in PRIMARY LOCALIZATION CUES beginning on page 13].

Application of spatialization depends upon hardware capabilities and fidelity requirements. For extremely high fidelity, all sounds, to include reflections, would be spatialized. For less accuracy, the original sound could be spatialized and reverb could be applied to approximate the environment. As fidelity requirements diminish or hardware capabilities become overtaxed, spatialization of sounds can be degraded.

B. THE APP-CULL-PLAY APPROACH

After investigating the families of modeling/rendering applications, an observation was noticed. Polygon modeling occurs before the real-time, graphics rendering process. A model is built from one or more polygons, or even from one or more other models. As these polygons and models are added, the graphical environment is constructed. The output of the modeling process is a file that describes the model or the environment.

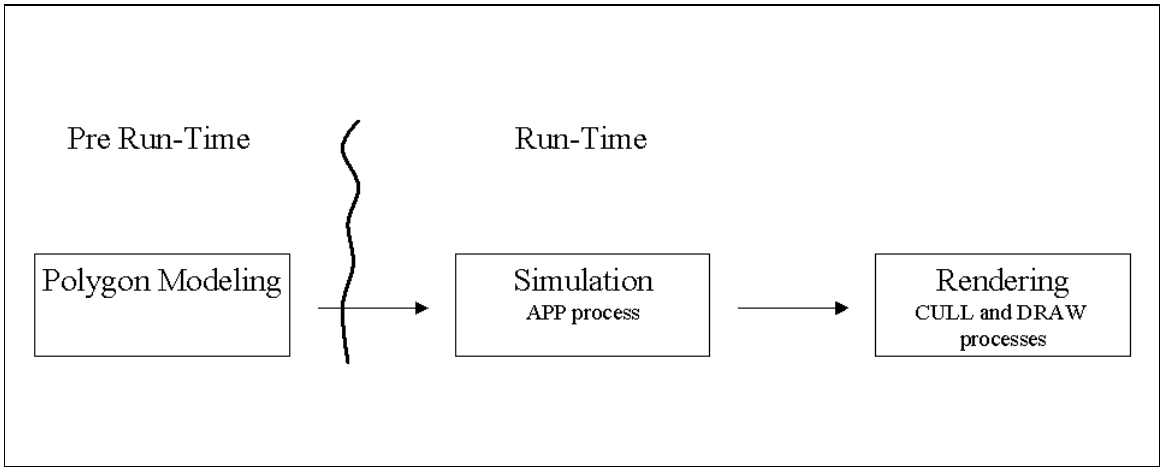


Figure 10: APP-CULL-DRAW process of graphical rendering.

The model's file is next imported into a real-time system. This system reads the model and develops the environment. This environment is then rendered for the user to see. This rendering program contains sections that run the application (real-time system), cull the polygons, and draw the polygons. This process is the APP-CULL-DRAW process, and it is depicted graphically in Figure 10.

Audio modeling also occurs as a step-time process, before the run-time environment can be run. Acoustical primitives, such as acoustical textures to reflect and to occlude sounds are added to an acoustical environment. As these acoustical characteristics are added, the auditory environment is constructed. The output of the modeling process is a file that describes either a sound or the acoustical properties of some object, called the audio geometry.

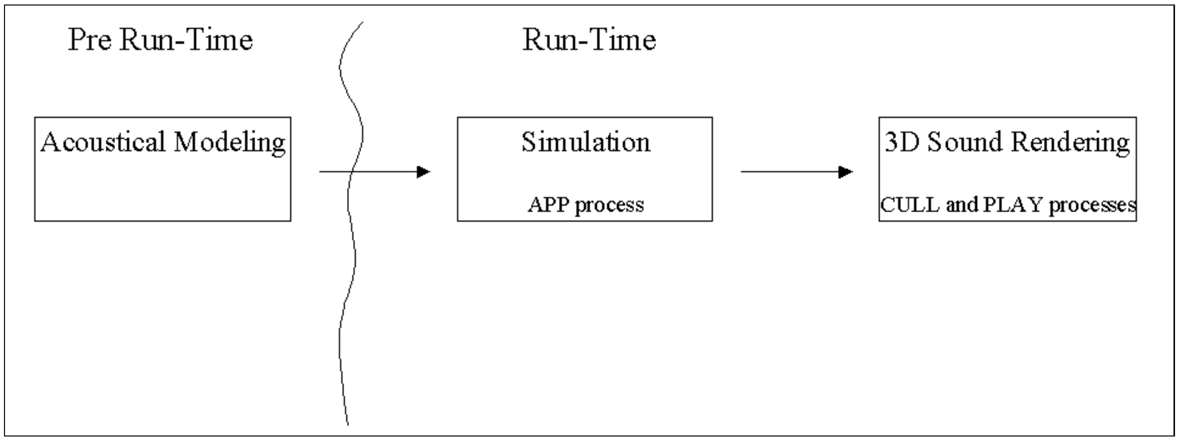


Figure 11: APP-CULL-PLAY process of audio rendering.

The audio model's file is next imported into a real-time system. This system reads the model and develops the audio environment. This audio simulation loads a specified environment and any necessary sounds. The audio environment is then rendered for the listener to hear. This audio rendering program contains sections that generate sounds (real-time system), cull all of the audible sounds, and play the sounds via speakers, headphones, or some other device. This process is the APP-CULL-PLAY process, and it is depicted graphically in Figure 11.

Figure 12 suggests a new method of developing and rendering environments. It combines both the visual and audio modeling process so that as one is being developed, the other is developed with little to no additional work. When the run-time system executes, it surveys both the graphical and the audio hardware, then builds the environment for whatever the given hardware will support. Finally, the APP-CULL-DRAW and APP-CULL-PLAY processes are integrated.

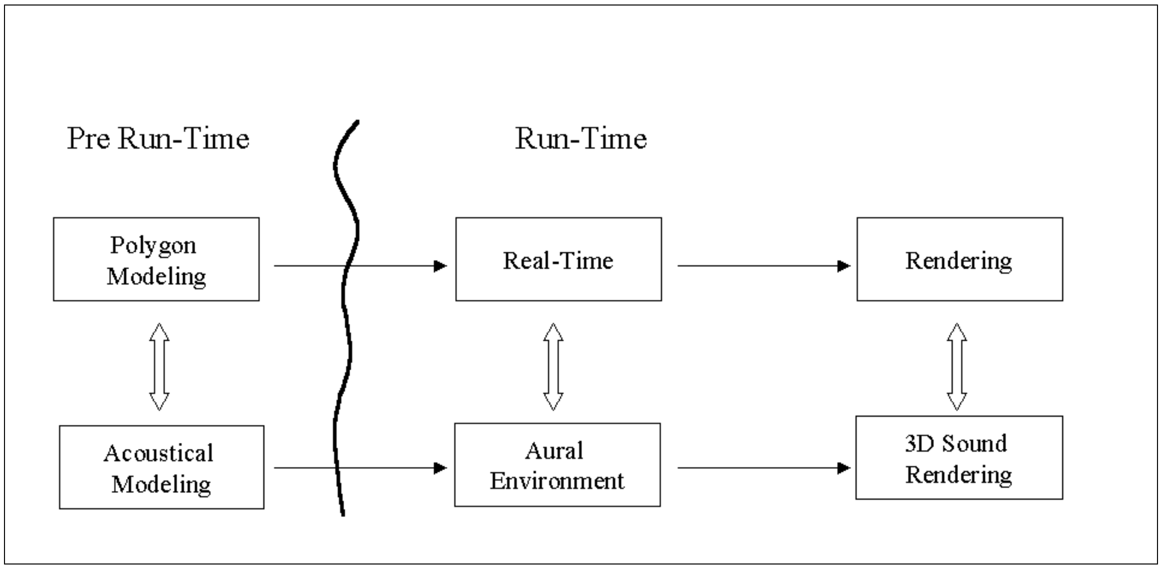


Figure 12: Integration of graphical and audio processes.

Another important aspect that was recognized is that the current APIs for addressing graphical hardware contain only elementary sound-related functions, specifically only functions for sound generation and reception (refer again to Figure 8 and Figure 9). Also noticed was the lack of graphical geometry support in the sound APIs. To remedy this, an overarching API is needed that addresses the graphics in a manner consistent with any competitive graphical API and addresses sound in all three of the sound phases. This should include the Audication and Aureflexion models detailed below.

C. AUDITORY MODELS

In graphics programming, many models exist. An illumination model expresses “the factors determining a surface’s color at a given point” [FOLE90]. The camera model allows us to specify the way the objects will be viewed. Texture mapping allows us to paste a “picture” onto a polygon to make the polygon look more real.

In the same manner, the audio environment needs models. A proposed set of models is discussed below.

1. Aureflexion (auditory reflection) model

The aureflexion model is used when simulating the acoustics of a room. This model does not render the location of an object in 3D space, per se. It uses the locations of all of the objects located in the three-dimensional space. It uses the acoustical properties of all of the objects in the three-dimensional space to calculate the actual sound that should reach each ear. This model requires an APP-CULL-PLAY process, and it is the PLAY portion of that process. This model uses the DSPs in the same manner that the graphical models use the graphical pipeline.

The aureflexion model is implemented in two ways. The first is acoustical ray tracing. Each object in the three-dimensional space is textured. The audio pipeline performs ray tracing to calculate the sound reflections as they should be received by the ear. This method relies heavily on DSPs or extremely fast main processors to perform something close to acoustical ray tracing.

The second method is a reverberation model. This is a close approximation to what the ear should hear. The acoustical properties of the three-dimensional space are calculated, and a loudness reduction is calculated. The acoustical properties are also used to calculate the sound decay of the room. Finally, the acoustical properties are used to calculate the reverberation time of the room. The reverberation time can be calculated for the full frequency spectrum, or it can be calculated for specific frequency bands for a more accurate implementation. The sound hardware would then apply the calculated reverb to the sound. This would approximate the reflected sound in a room.

2. Audication (Auditory location) Model

The audication model is used to represent the location of a sound in the three-dimensional space. The audication model uses HRTF, IID/ITD, or any other acceptable method to represent the sound in a specific location. The Auditory location model can be used in conjunction with the aureflexion model, but this is not a requirement.

Figure 13 demonstrates the relationships between the two proposed audio models and the three states of a sound, as detailed on page 33. Notice that no model is presented for the generation state; that is beyond the scope of this thesis.

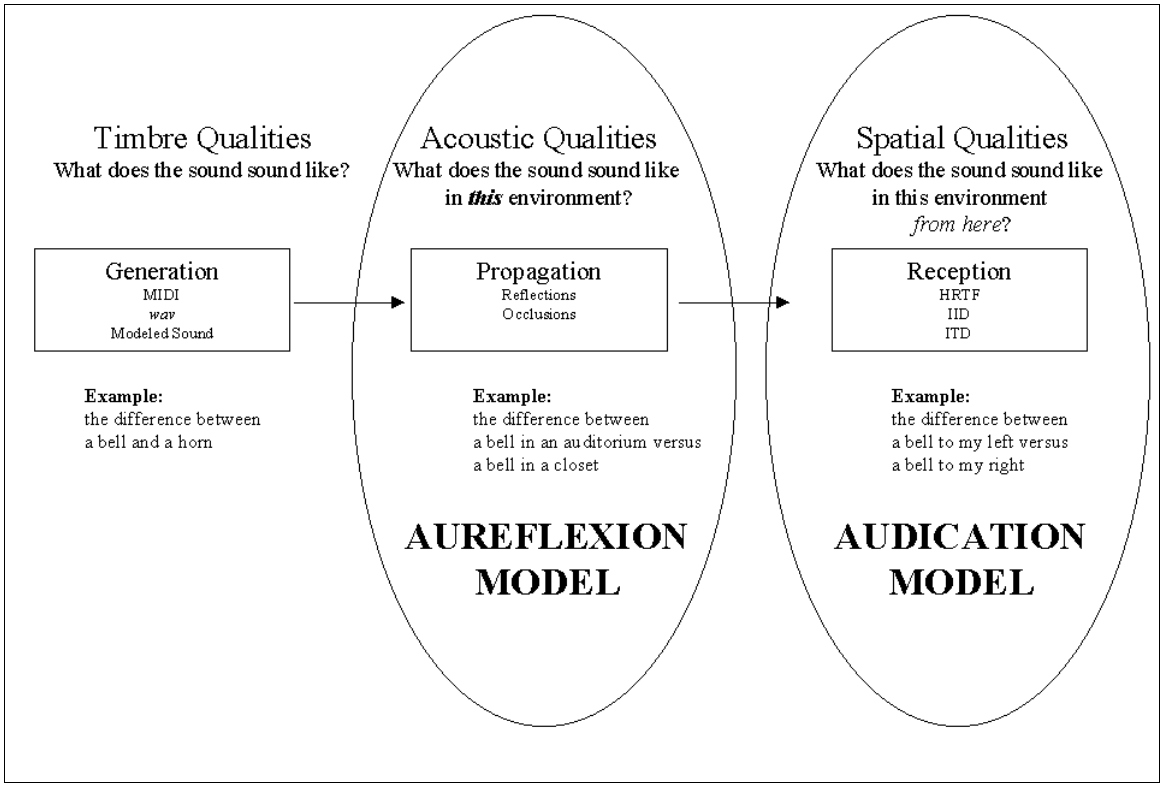

Timbre Qualities
What does the sound sound like?
Acoustic Qualities
What does the sound sound like in *this* environment?
Spatial Qualities
What does the sound sound like in this environment *from here*?
Generation
MIDI
wav
Modeled Sound
Propagation
Reflections
Occlusions
Reception
HRTF
IID
ITD
Example: the difference between a bell and a horn
Example: the difference between a bell in an auditorium versus a bell in a closet
Example: the difference between a bell to my left versus a bell to my right
AUREFLEXION MODEL
AUDICATION MODEL

Figure 13: Relationships between Audio Models and the three states of sound.

D. CONCLUSION

This approach provides an opportunity to simplify the tasks of the modeler and the programmer. First, by combining the model building, the modeler needs expertise in either graphics or audio, but is not required to be an expert in both. Second, since the environment builder is not required to build for a specific set of hardware, concentration is placed on the environment, not the machinery that will run the environment. Finally, by marrying the geometrical and auditory programming calls, redundancy is reduced and consequently the likelihood for errors is reduced.

V. IMPLEMENTATION

Chapter IV described the integration of the visual and audio portions of virtual environments that should occur in a production application. For this thesis, however, only a proof of concept will be implemented. As proof of concept, it was decided to implement a modeling tool capable of building both the visual environment (geometry) and the auditory environment (sound and geometry) in an integrated manner. The modeler must at a minimum be able to create a room, color the room, add a texture to the walls of the room, add a single sound, and move that sound within the room. The modeler must be run on a machine that can render simplistic geometric primitives with textures, and render three-dimensional sound with the ability to texture the sound.

Once it is proven that the environment can be easily developed using this visual/auditory modeler, it must be proven that such a model can be easily run with or without sound in a simple rendering program. The rendering program should load the visual geometry, the audio geometry, or both. It should load other models that may or may not have sounds attached to them. Once the models (visual and audio) have been loaded, the renderer should allow a user to peruse the environment. As the user moves through the environment, the audio display and the visual display should change appropriately.

To accomplish the above, a cursory review of hardware for both sound display devices and graphical rendering machines was accomplished. In reviewing the hardware, consideration was given to both technical issues (which device is best), resource issues (a machine's rendering capability vs. its availability), and longevity issues (will this be around two years from now?).

A. SELECTING HARDWARE

To render the graphics, there was a choice of three basic systems. The IBM PC systems available had the ability to render some graphics, but were not outfitted with strong 3D Graphics cards. There were also only three PCs available to be shared among more than a half dozen students, so availability was questionable. The PCs were outfitted with SoundBlaster AWE 32 sound cards. These sound cards were capable of rendering

3D audio, but audio texturing was not available on the sound cards. Although the PC showed much potential for future work, it was determined that anything developed should be applicable for both the PC and SGI hardware. For these reasons, the PC systems were less than desirable for this project.

The available SGI computers had strong rendering capability; all were sufficient for a small rendering project. Only the SGI Onyx Infinite Reality (IR) workstations are capable of rendering very complex textured geometry. Since these had the most rendering power, they were heavily used by most of the students. An SGI Indigo2 was available and capable of running textured graphics on a smaller scale than the IR workstations; consequently, it was chosen for graphics rendering.

This machine has the following hardware:

- Iris Audio Processor: version A2 revision 0.1.0
- 1 100 MHZ IP22 Processor
- FPU: MIPS R4000 Floating Point Coprocessor Revision: 0.0
- CPU: MIPS R4000 Processor Chip Revision: 3.0
- On-board serial ports: 2
- On-board bi-directional parallel port
- Data cache size: 8 Kbytes
- Instruction cache size: 8 Kbytes
- Secondary unified instruction/data cache size: 1 Mbyte on Processor 0
- Main memory size: 64 Mbytes
- EISA bus: adapter 0
- Integral Ethernet: ec0, version 1
- Integral SCSI controller 1: Version WD33C93B, revision D
- CDROM: unit 4 on SCSI controller 1
- Integral SCSI controller 0: Version WD33C93B, revision D
- Disk drive: unit 1 on SCSI controller 0
- Graphics board: GU1-Extreme

For acoustical rendering, the SGI machines were not as strong as needed. The SGI machines had the processing power to calculate the correct sounds in both HRTF and reflections, but this would require developing libraries that would calculate the sounds. The actual coding of these functions is not within the scope of this thesis.

The Acoustetron II can render up to sixteen 3D sounds, or four 3D sounds with room acoustics. It comes with an API that allows easy development of code to interact with it. It was also unused, giving easy access for this research. The Acoustetron II has the following characteristics:

- 4 DSP signal-processing cards
- Acoustetron II Client Software Library and Demos
- 8 concurrent 3D sources at 44kHz sample rate
- 16 concurrent 3D sources at 22kHz sample rate
- 4 concurrent 3D sources with 6 reflections at 44kHz sample rate
- pitch shift control for all sources
- Maximum system update rate: 44Hz

B. SELECTING SOFTWARE

1. Acoustical Server

The Acoustetron II came with a software base and demo software. It was decided to develop an audio server to communicate between the graphical and audio portions of the project. Initially development began on an audio server that communicated via sockets with the main application. The acoustical server is based heavily upon CRE's audioClient program and the CRE_TRON Function Reference. [ACOU96] The acoustical server is named AcousteLib.

2. Modeling Tool

Open Inventor [WERN94] [WERN93] was chosen for the modeler portion of this thesis [depicted as Pre Run-Time in Figure 12 on page 37]. Inventor includes many manipulators, like transforms, transformManipBox, etc. that allowed easy development of a modeling tool. Also, Inventor is a very useful 3D API, so learning it could provide more insight into Visual Graphics.

The demonstration program *Linkatron* allowed a user to manipulate models as they moved down a plane. The source code provided a good starting point for learning Inventor and the C++ bindings. *Linkatron*, however, did not contain the appropriate code-base for extending into a modeling tool.

Further digging uncovered two programs, *SceneViewer* and *Gview*. Both of these demonstration programs shipped with source code. *SceneViewer* addressed the reading and writing of files, but *Gview* represented the model in two windows; a visual window and a scene graph window. Both of these programs could serve as a base modeling tool, to be extended for this thesis' needs. After evaluating each program from a user's perspective, *Gview* was chosen as the base for this thesis' modeling tool. It was renamed to *GAUDIVIEW* and the code was extended extensively.

3. Rendering Tool

Iris Performer [depicted as Run-Time in Figure 12 on page 37] was chosen for the database rendering application. Performer ships with the demonstration application *Perfly*. Having used *Perfly* as a base for a previous project, familiarity was not an issue. *Perfly* was already a working database rendering application, so only the sound portion needed to be implemented. Once audio capabilities were added, it was renamed *GAUDIFLY*.

C. TYPEDEFS

During the development of AcousteLib, it was determined that a collection of type definitions would allow all programs to share data types. The file is named *typedefs.h*. For easy access, it is located in the home directory.

After a great deal of thought, a few data types were determined to be necessary for this project. This is not an all-inclusive list of data types, only those necessary for this proof of concept. These type definitions are represented throughout this section in many tables. Further proposed type definitions are shown in Chapter I, Section C.

Representing a room, or a listening environment, is the most critical of the three data structures in this project. The data types naturally group themselves into five groupings. The first of these groups gives us the room ID and its location. The room ID is necessary to address the Acoustetron II. The second group gives us the location of the room. The room has X, Y, and Z coordinates, and a size for the X, Y, and Z. This is implemented in this fashion to address the Acoustetron II.

Since a room is made of six walls, the next grouping represents the material covering the wall. This information is represented by six integers, wall1material – wall6material. Although the Acoustetron II can apply only one material to all walls, it was decided to represent all six walls in this data structure for consistency. The integer represents one of the materials the Acoustetron II can apply to a wall, as depicted in Table 4.

Integer Value	Material	Graphical Texture File
0	Anechoic	Anechoic.rgb
1	Mirror	mirror.rgb
2	Textile	Carpet.rgb
3	Plywood	wood.rgb
4	Glass	Glass.rgb

Table 4: Acoustical Materials

In addition to applying a texture, or wall material, to each of the walls, the Acoustetron II supports a loudness factor for each wall. In this manner, four walls could reflect sound according to the default texture, while the front and back walls were set to a loudness of –120dB. This would effectively allow us to remove walls from the acoustical environment, if that were necessary. To accommodate this, six integers for wall loudness were included. [this relates to *liveness* as discussed on page 15]

```
typedef struct {
      int   roomID;
      float roomX; // lower left corner of room
      float roomY;
      float roomZ;

      float roomSizeX; // width of room
      float roomSizeY; // height of room
      float roomSizeZ; // depth of room

      int   wall1material;
      int   wall2material;
      int   wall3material;
      int   wall4material;
      int   wall5material; // floor
      int   wall6material; // ceiling

      int   wall1loudness;
      int   wall2loudness;
      int   wall3loudness;
      int   wall4loudness;
      int   wall5loudness;
      int   wall6loudness;

      float roomCoef128;
      float roomCoef256;
      float roomCoef512;
      float roomCoef1024;
      float roomCoef2048;
      float roomCoef4096;

} roomStruct;
```

Table 5: Room structure

The final grouping of data for the room acoustics contains the room coefficients. Since the room coefficients are variable, depending upon the objects inside the room, these coefficients were added so the total room coefficients could be calculated and stored. Since the Acoustetron II did not support this function well, these were not used.

The final data structure for the room is shown in Table 5.

Representing a head, or a listener, was accomplished with an *ID* and a float array with six members. The *ID* is used to determine which listener is being referenced, in case the scene contains more than one user with more than one simultaneous viewer/listener. The six floats represent the listener's X, Y, and Z coordinates and the listener's heading, pitch, and roll. The head structure is shown in Table 6.

```
typedef struct{
      int     ID;
      float   location[6];
} headStruct;
```

Table 6: Head structure

Representing a sound is rather straightforward. The sound structure, named *soundVector*, consists of a *soundID*, a file name, a pointer to the *wav*, a location, and an amplification factor, as shown in Table 7. The *soundID* represents the specific sound for the Acoustetron II. When applying any function to a sound on the Acoustetron II, the *soundID* is required. The variable *fname* represents the actual name of the *wav* file. The *wav_ptr* is an Acoustetron II data structure and is used to track the sound. The sound *location* is represented by a float array. The variable *amplifyDB* is used to adjust the loudness of the sound to compensate for inequalities in the audio level in sound recordings.

```
typedef struct {
      int     soundID;
      char    fname[13]; // = “welcome.wav”; //char    *sPtr;
      wavFt   *wave_ptr;
      float   location[6];
      float   amplifyDB;
} soundVector;
```

Table 7: Sound structure

For the purpose of testing and early development, an overarching data structure was built. It contained enough information to represent a room, a head, and four sounds – the maximum number of sounds the Acoustetron II can render when performing room acoustics. This data structure is represented in Table 8.

```
typedef struct {
      roomStruct  theRoom;
      headStruct  theHead;
      soundVector theSound0;
      soundVector theSound1;
      soundVector theSound2;
      soundVector theSound3;
} audioStruct;
```

Table 8: Preliminary Audio Structure

D. ACOUSTELIB

The Acoustetron II came with some example programs and a library to allow development of applications. The function calls appeared to be well thought out, and laid out in a logical order. It was determined that to implement this project in a most efficient manner, function calls were needed that could be easily incorporated into any program, such as initializing the sound device, playing a sound, etc. The Acoustetron library had a function to initialize the sound device. It also had a function to initialize a listener. It contained a function to load a sound, a function to play a sound, a function to apply volume to a sound, etc. To make programming easier, a single function to play a sound, like *playSound*, should load the sound, set its volume, and start the sound playing.

After it was decided to implement this as an API, development became straightforward. Routines were implemented to initialize the sound device and to close it. Routines were implemented to load a sound, play a sound, stop a sound, and update a sound location. A function to update the listener's location was also implemented. Finally, a routine called *setupAcousteTron* was implemented. This routine, which was not yet complete, could be called to implement all other routines needed to get the sound device up and running with sound. At this point, each routine made its own call to *cre_update_audio().*

It was decided to create another function called *afFrame()*. This is to mirror the Performer function *pfFrame()*. *afFrame* tells the sound device to apply any changes that it has received since the last *afFrame*. This allows the programmer to control update or frame rates, and allows the programmer more control over the control loop. The programmer can also program the *afFrame* immediately after the *pfFrame*. This allows the audio and video to move at the same time.

At this point, it was decided to write functions to read and write an audio environment. The information needed to represent an acoustical environment is addressed in the section on Typedefs. Reading and writing the acoustical environment is accomplished with the C++ statements *cin* and *cout*.

```
int initAcousteTron(headStruct head);
audioStruct setupAcousteTron(audioStruct audioWorld);
void closeAcousteTron();
void updateSoundLocation(soundVector sound);
soundVector updateSoundLocation(soundVector sound,
                                vectorStruct PerflyData);
void updateListenerLocation(headStruct head);
headStruct updateListenerLocation(headStruct head,
                                  vectorStruct PerflyData);
soundVector loadSound(soundVector sound);
soundVector closeSound(soundVector sound);
void playSound(soundVector sound); //
void playSoundOnce(soundVector sound); //
void stopSound(soundVector sound);
void amplifySound(soundVector sound);
soundVector amplifySound(soundVector sound, float value);
roomStruct makeARoom(roomStruct room);
void updateRoom(roomStruct room);
void textureRoom(roomStruct room);
roomStruct textureRoom(roomStruct room, int material);
roomStruct removeTexture(roomStruct room);
void adjustWallLoudness(roomStruct room);
void saveAudioGeometry(audioStruct audioWorld);
void saveRoom(roomStruct theRoom);
void saveRoom(roomStruct theRoom, char *filename);
void saveSound(soundVector theSound);
void saveSound(soundVector theSound, char *filename);
audioStruct readAudioGeometry();
audioStruct readAudioGeometry(char *filename);
roomStruct readRoom(char *filename);
soundVector readSound(char *filename);
int defineOutput(int device);
void afFrame();
```

Table 9: Functions implemented in Acoustelib

After many iterations of compile, execute, debug and test, the necessary function calls were implemented. These functions are shown in Table 9. These functions represent a hardware dependent implementation of an acoustical server. Since this thesis required only a proof of concept on only one set of hardware, the functions were hardware specific. These functions represent the minimum necessary functions to implement sound generation, propagation, and reception.

A small test program, *acoustelibTest*, was developed to test the new AcousteLib. It first instantiated the *audioStruct* data structure. Throughout the many versions of *acoustelibTest*, each of the function calls was tested.

E. GAUDIVIEW

The first task in implementing GAUDIVIEW was to load an Inventor file. The original program would load an Inventor file from the command line, but once the program had begun, no other models could be imported. Although this was not technically necessary, it was determined that a modeler that could not import any pre-built models was a little too simplistic. This was not a proof-of-concept issue, it was a simple pride issue.

```
SbBool openFileStuff(SoInput *in)
{
    SbBool ok = TRUE;
    int argc = 0;
    char * * argv;

    argv = new char*[1];
    argv[1] = new char[1];
    argv[1][1] = ‘\0’;

    VkApp *scaleApp = new VkApp(“GetFileStuff”, &argc, argv);

    SbBool value;

    VkFileSelectionDialog(myTitle);

    // SET THE FILTER TO *.IV
    theFileSelectionDialog->setFilterPattern(“*.iv”);
    theFileSelectionDialog->setDirectory(“.”);

    value = theFileSelectionDialog->postAndWait( );
    if (value == VkDialogManager::OK)
      { ok = in->openFile(theFileSelectionDialog->fileName()); };

    if (value == VkDialogManager::CANCEL)
      { cout << endl << “I don’t know what happened” << endl; };

    return ok;
}
```

Table 10: File Dialog Code

Loading an Inventor file was actually a rather easy task. A *SoInput* node was instantiated with the name *&inFile*, then assigned to the file. A simple code snippet to read an Inventor file was assigned to the *bufferNode*, a *SoNode* type. The original program came with a Paste routine. Once the file was read, it was assigned to a *bufferNode*. Next, the Paste routine was called. Now the newly read Inventor file could be pasted into the scene graph. This is shown in Table 10.

The second implementation task was to build a room. Since this modeler was intended to model room acoustics in addition to graphics, this task was to be the cornerstone of the modeler. Still not being overly familiar with Inventor, outright creating a *SoBox* node and inserting it into the scene graph did not seem like the easy way to go. Using the current executable version of *Gview*, a simple cube was built and saved in a file named *room.iv*. The original code for loading an Inventor file was resurrected, and the filename was hard-coded. This seemed like the easiest method of building the room. This code is shown in Table 11.

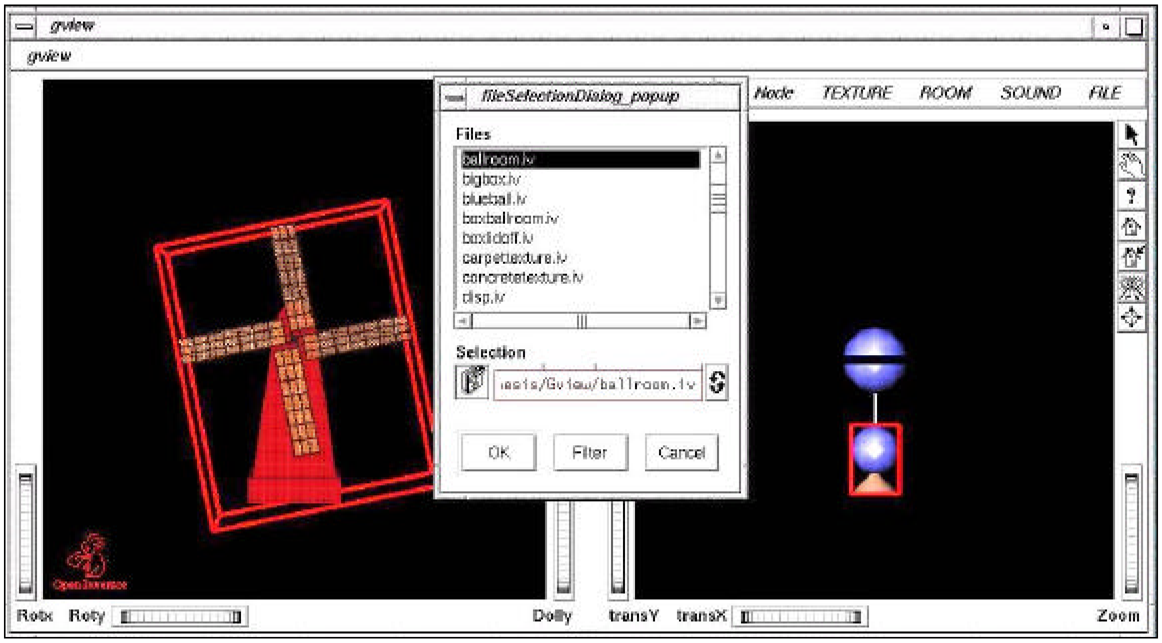


Figure 14: File Selection Box..
The file 'ballroom.iv' is being selected. A windmill has already been introduced into the scene.

To represent a room made of four wooden walls, we not only needed to acoustically build the room, but also to build the graphical portion of the room. Now that a room could be built, or at least loaded, texturing the room became the next priority. Texturing a wall is necessary to accomplish sound diffusion.

It was determined that the easiest way to accomplish this would be to create a texture node and save it to an Inventor file. A file could be created for a wood texture, a material texture, and a glass texture. Then any of these could be loaded. By reusing the code snippet to load the file again, this task could be easily accomplished.

```
case ROOM_INSERT:
            // CREATE THE ROOM

            // OPEN THE FILE FOR READ
      ok = oFile(“room.iv”, &inFile);

      if (ok)
        {
            // SET THE bufferNode to ‘the input file’
          bufferNode = SoDB::readAll(&inFile);

            // CLOSE THE FILE
          inFile.closeFile();
      }
      else {}; // do nothing

      if (bufferNode != NULL)
      {
            // PLACE IT IN THE SCENEGRAPH
          pasteByRef = FALSE;
          pasteBegin(ecb);

            // PLACE IT ON THE ACOUSTIC LANDSCAPE
          audioWorld.theRoom.roomID = 0;
          audioWorld.theRoom.roomX = 0;
          audioWorld.theRoom.roomY = 0;
          audioWorld.theRoom.roomZ = 0;
          audioWorld.theRoom.roomSizeX = 1;
          audioWorld.theRoom.roomSizeY = 1;
          audioWorld.theRoom.roomSizeZ = 1;

          audioWorld.theRoom =
               makeARoom(audioWorld.theRoom);

          afFrame();
      }
      else {}; // do nothing

      break;
```

Table 11: Code to create a room

At this point, the method of loading a predefined Inventor file had reached the end of its utility. The modeler could create a room, and that room could be textured. The modeler could keep track of the texture with a variable. But what if the user wanted to change textures? Although we could just add another texture (the rightmost texture in the scene graph is the one that the user will see), this was no longer an easy workaround. It became time to find a new technique.

Since any node in the scene graph can be named, naming seemed the appropriate method. The room would be named *RoomNode*. The texture node would be named *RoomTexture*, and any other unforeseen nodes will be named as necessary. Once these nodes are named, we can simply traverse the scene graph and perform operations to any nodes. This seemed like **real** graphics programming, so it was the next endeavor undertaken.

Searching the scene graph was as easy as originally thought. Finding a node named *RoomTexture* was accomplished with the following single line of code:

```
bufferNode = bufferNode->getByName("RoomTexture");
```

Actually applying the texture was also easy. Using the functions inherited from the Inventor *SoField* node, changing the texture in a pre-existing texture node was accomplished by the code in Table 12. A room textured with wood is shown in Figure 15.

```
SoField *field = bufferNode->getField("filename");
field->get(string);
if (texture == CARPET_INT)
    field->set(CARPET);
```

Table 12: Modifying a texture

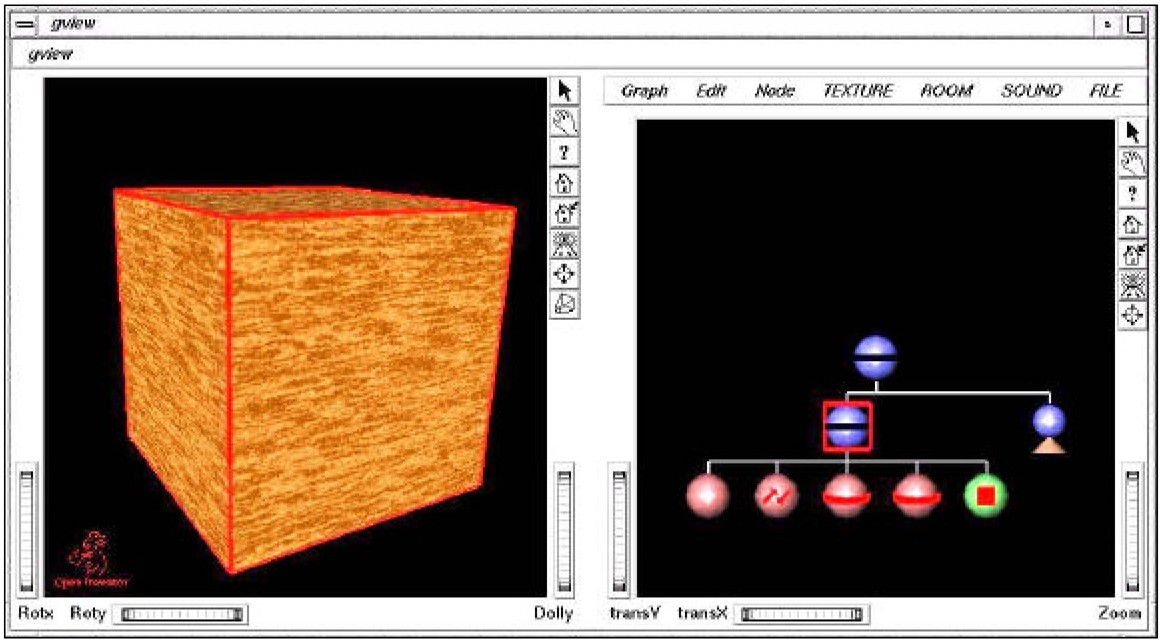


Figure 15: GAUDIEVIEW showing a simple room textured with wood. The left portion of the screen shows the display graph. The right portion of the screen displays the scene graph.

Now the modeler could create a room and texture it. Since this is the barebones minimum graphical modeling ability required to show proof of concept, it was time to add sound to the modeler. The data structure *audioStruct* was instantiated with the name *audioWorld*, and the makefile was altered to include *AcoustLib.a*. This was completed easily, so it was time to begin implementing sound into GAUDIVIEW.To initialize the Acoustetron II, a head was created and placed in the scenegraph at the origin. The Acoustetron was then initialized. Finally, the head location was initialized on the Acoustetron. The head represents the location that the modeler's "ears" would be in the model.

A few pre-recorded sounds were added to the project. The project would have the ability to reproduce a voice saying "Hey" and another saying "Test one two". These sounds were chosen because these are two methods used when someone wants to acoustically size a room. A trumpet sample of Maynard Ferguson playing the introduction to *Gospel John* [FERG74] was the third sound incorporated. This was used to simulate musical sounds played in a room. The fourth sound was the sound of a helicopter, specifically the sound file *4heli1.wav* from the Acoustetron II's sound library. Figure 16 shows a head node along side a sound node that has been selected.

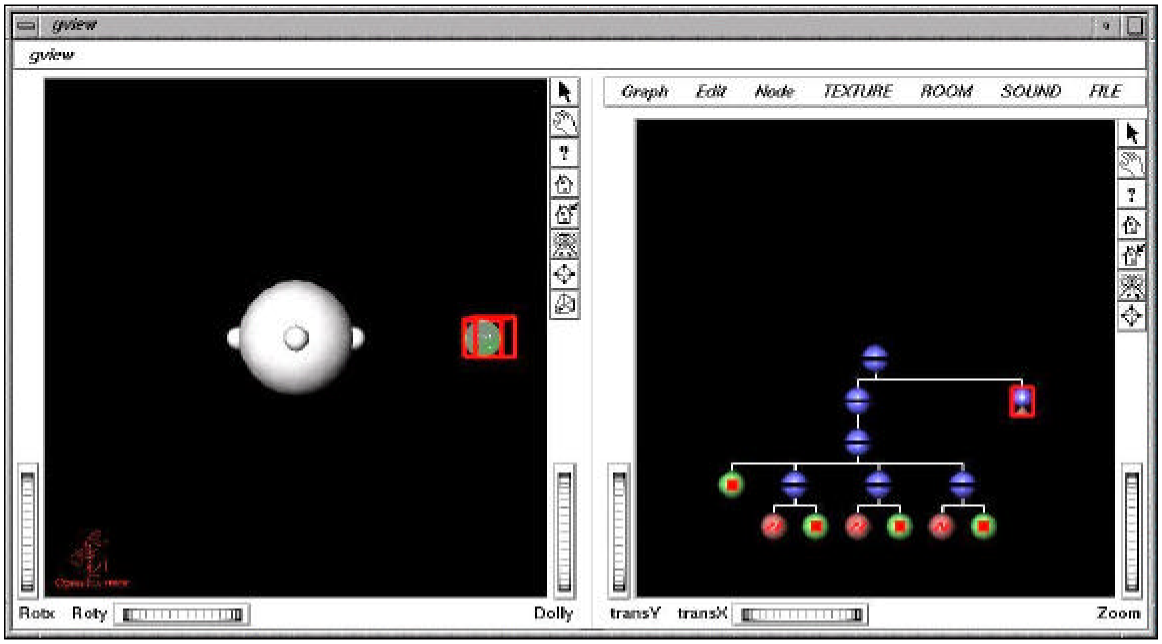


Figure 16: Head node and Sound node.
The Head node on the left and the "Hey" node, selected, are shown in the display graph portion of GAUDIVIEW.

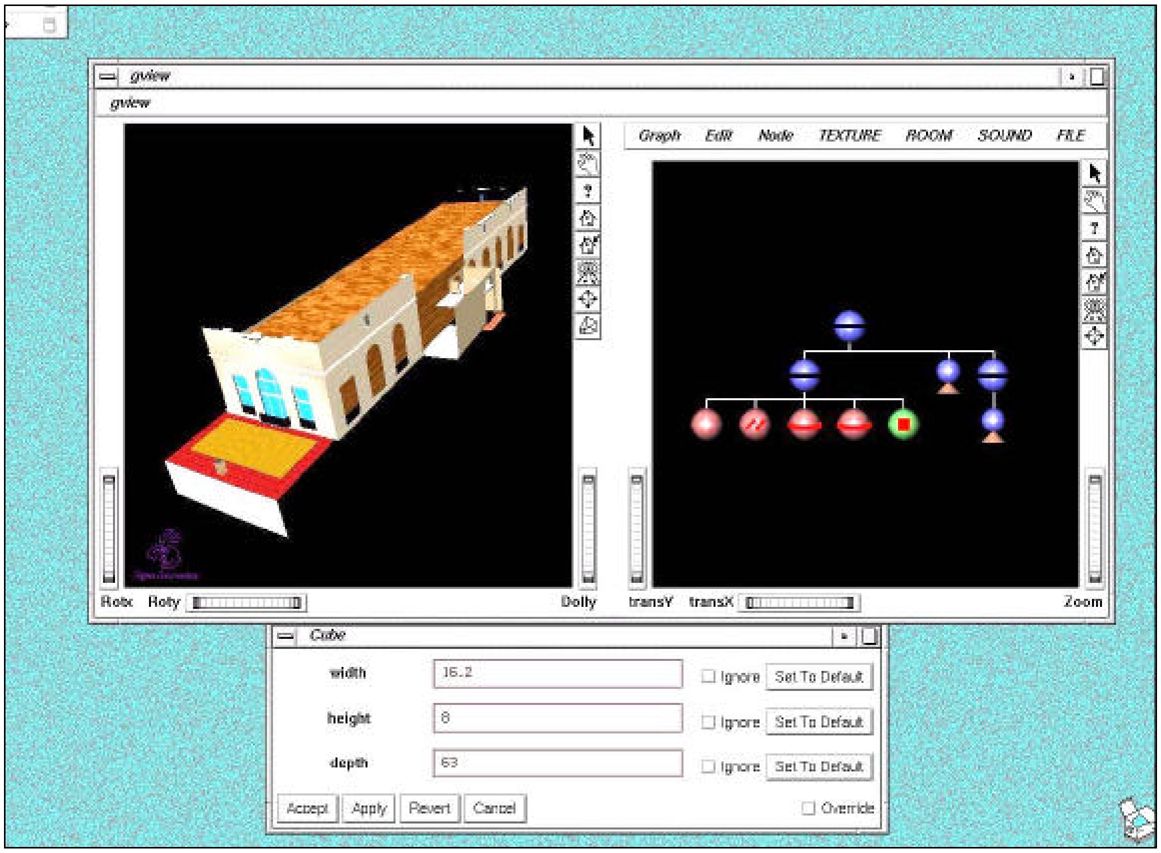
Actually implementing the sounds into the modeler was accomplished in much the same manner as initializing the sound device. First, an Inventor file containing the appropriate graphic was loaded, and then it was pasted into the scene graph. Next, the sound portion of the data structure *audioWorld* was initialized and the sound was loaded. The sound location and amplification were updated. Finally, the Acoustetron was told to play the sound. This code is shown in Table 13. All of the sounds were implemented in much the same manner.

```
case SOUND_HEY:

          // IF SOUND IS NOT ALREADY LOADED
      if (hey == -1)
      {
          // OPEN THE FILE FOR READ
          ok = oFile(“hey.iv”, &inFile);

          if (ok)
          {
               // SET THE bufferNode to ‘the input file’
            bufferNode = SoDB::readAll(&inFile);

                // CLOSE THE FILE
            inFile.closeFile();
          }
          else {}; // do nothing

          // PLACE IT IN THE SCENEGRAPH
          pasteByRef = FALSE;
          pasteBegin(ecb);

            // DEFINE THE SOUND
          audioWorld.theSound2.soundID = 2;
          strcpy(audioWorld.theSound2.fname,
                “hey.wav”);
          audioWorld.theSound2 =
                loadSound(audioWorld.theSound2);

            // GIVE IT A LOCATION AT THE ORIGIN +5 +5 0
          audioWorld.theSound2.location[0] = 1.0;
          audioWorld.theSound2.location[1] = 1.0;
          audioWorld.theSound2.location[2] = 0.0;
          audioWorld.theSound2.location[3] = 0.0;
          audioWorld.theSound2.location[4] = 0.0;
          audioWorld.theSound2.location[5] = 0.0;

          updateSoundLocation(audioWorld.theSound2);


            // GIVE IT SOME VOLUME
          audioWorld.theSound2.amplifyDB = 100.0;
          amplifySound(audioWorld.theSound2);


            // PLAY THE SOUND
          playSoundOnce(audioWorld.theSound2);

            // INCREMENT helo
          hey = 1;
      }

      else if (hey >= 0)
           {
                      // PLAY THE SOUND
                playSoundOnce(audioWorld.theSound2);
                hey = 1;
           }
           else {} // DO NOTHING
      break;
```

Table 13: Loading a sound in GAUDIVIEW

Now that the sounds were playing, we needed to be able to adjust parameters of the room to hear the differences that the adjustments made. This was perhaps the most difficult portion of the Inventor coding for this thesis.

To resize the room, the room must first be located within the scene graph. This was done in the same manner as finding a texture node. Once the room was found, getting the height, width, and length data from the cube node was not very intuitive. After much research through numerous *man* pages, the code snippet in Table 14 was found and modified. By placing it into a function call, *getFieldValue* could be called with a generic node and the name of a field, and it would return the value of that field. This was important because by retrieving the value held in the field, the benefits of Inventor and *Gview*, the modeling program that GAUDIVIEW was extended from, could be realized. Since code already existed to modify the attributes of any node, the room could be resized by simply double-clicking on it. An example of resizing a room is shown in Figure 17.



width	16.2
height	8
depth	63

Figure 17: GAUDIVIEW in action.
The model of Herrmann Hall Ballroom is courtesy of John Locke, NPSNET Research Group member. The cube represents the acoustical "room". It is being sized to match the graphical model of the Ballroom.

After the new values were applied, the user could select an option called *Apply Changes*. This function walked through all of the nodes looking for audio information. The audio structure was updated with the data from these nodes. The room's size was determined and set. The room's location was determined from the *RoomTransform*, then set on the Acoustetron. The room texture was determined, and its value applied to the room's audio material variable.

```
float getFieldValue(SoNode *bufferNode, char *fieldname)
{
    float returnFloat;
    SoField *field = bufferNode->getField(fieldname);
if (field->isOfType(SoSFFloat::getClassTypeId()))
        {
          SoSFFloat *floatField = ((SoSFFloat *)field);
          returnFloat = floatField->getValue();
        }
return returnFloat;
}
```

Table 14: Code to get the value in a specified field

At this point, GAUDIVIEW was modeling both the sound and the geometry successfully. It was now time to save the audio environment. Since a function to save an audio environment was written and tested in AcousteLib, a simple call to the procedure saveRoom accomplished this task.

Now that visual and auditory models could be saved, it was time to clean up the loose ends. In the same vein that the audio environment was updated, the sounds could be updated. A sound could be moved by selecting the sound's transform node and adjusting its values. By adding a few more lines of code to the Update Audio function, translation values for each of the sounds were obtained from the scene graph and applied to the Acoustetron II.

For the purposes of this demo, it was determined that the damping factors of the walls should be adjustable. This would allow a user to make slight modifications to the "material" that covered the walls, and would demonstrate that these parameters should be adjustable. A function was added to adjust the liveness of the floor, ceiling, and each of the walls.

Finally, a function was added to select the audio output medium. Since the Acoustetron II supported output to headphones, or speaker systems, this was also included in the modeler.

F. GAUDIFLY

Since *Perfly* was already a working renderer, it only needed integration of the sound portion. This was done by adding only a few lines of code. Immediately after *Perfly* executed the function *pfFrame()*, the acoustical function *afFrame()* was added. When the viewpoint moved, the hearing point was also moved.

```
      // IMPLEMENT THE AUDIO STRUCTURES
    roomStruct          theRoom;
    headStruct          theHead;
    soundVector         theSound[NUMSOURCES];
    int                 numSounds;
    int                 AUDIODEVICELOADED;
    int                 SPEAKERTYPE;
    int                 SOUNDON;
```

Table 15: Declaring the audio structures in GAUDIFLY

To implement sound into *Perfly*, creating GAUDIFLY, the sound devices needed to be declared. These were added to the *ViewState* structure already in existence. Unlike GAUDIVIEW, the test structure *audioWorld* was not used; instead, each of the structures was implemented in its own right. This is shown in Table 15.

To align the viewer's hearing location with the current camera's view, the location variables for the current head were assigned the values of the *ViewState*'s *viewCoords*. This is shown in Table 16.

```
ViewState->theHead.location[0] = ViewState->viewCoord.xyz[0];
ViewState->theHead.location[1] = ViewState->viewCoord.xyz[1];
ViewState->theHead.location[2] = ViewState->viewCoord.xyz[2];
ViewState->theHead.location[3] = ViewState->viewCoord.hpr[0];
ViewState->theHead.location[4] = ViewState->viewCoord.hpr[1];
ViewState->theHead.location[5] = ViewState->viewCoord.hpr[2];
updateListenerLocation(ViewState->theHead);
```

Table 16: Applying Viewer's Coordinates to the sound device

Once GAUDIFLY was working correctly using *Acoustelib*'s *initSound()*, which loaded a pre-working acoustical environment, it was time to start initializing the sound device by specific function calls, and loading environments and sounds from the command line. The command line parameters are shown in Table 17.

Command Line Parameter	**Effect of parameter**
-A	Initialize the sound device
-S	Audio environment to load
-s	Sound file to load

Table 17: Command line parameters for AUDIFLY

The command line parameters were implemented in the file *cmdline.C* as shown in Table 18. The option *A* was modified to initialize the sound device. The variable *theHead* was initialized and then the *AcousteLib* function *updateListenerLocation* was called. The option *S* was modified to load an audio environment. The option *s* was modified to load an audio sound.

Once these command line parameters were implemented, GAUDIFLY specifically met the requirements set forth for this project. It could load and run a sound free virtual environment. It could load and run an environment with sounds. It could load and render a fully specified acoustical environment. Figure 18 shows GAUDIFLY rendering a helicopter and Herrmann Hall.

```
// INITIALIZE THE SOUND DEVICE
      case ‘A’:
                  // GET THE AUDIO FILE NAME
            ViewState->theHead.location[0]
                              = ViewState->viewCoord.xyz[0];
            ViewState->theHead.location[1]
                              = ViewState->viewCoord.xyz[1];
            ViewState->theHead.location[2]
                              = ViewState->viewCoord.xyz[2];
            ViewState->theHead.location[3]
                              = ViewState->viewCoord.hpr[0];
            ViewState->theHead.location[4]
                              = ViewState->viewCoord.hpr[1];
            ViewState->theHead.location[5]
                              = ViewState->viewCoord.hpr[2];
            updateListenerLocation(ViewState->theHead);

                  // SETUP THE ACOUSTETRON
            if (initAcousteTron(ViewState->theHead) == 0)
            {
                ViewState->AUDIODEVICELOADED = 0;
            }
            else
            {
                ViewState->AUDIODEVICELOADED = 1;
            }

          break;

      // READ THE ROOM SOUND AND SET UP THE ROOM (ACOUSTICALLY)
      case ‘S’:

          ViewState->theRoom.roomID = -1;
          ViewState->theRoom = readRoom(optarg);
          if (ViewState->theRoom.roomID == -1)
          {
            usage(“BAD ROOM FILE NAME”);
          }
          else
          {
            // SET UP THE ROOM
            ViewState->theRoom.roomID = 0;
            ViewState->theRoom =
                makeARoom(ViewState->theRoom);
          }
          break;

      // READ A SOUND FILE, ADD IT TO THE SCENE AND THE ACOUSTICS
      case ‘s’:
            ViewState->theSound[ViewState->numSounds]
                  = readSound(optarg);
                  if (ViewState->theSound[ViewState->numSounds].soundID
                              == -1)
            { usage(“BAD .snd FILE”); }
            else
            {
                ViewState->theSound[ViewState->numSounds]
```

```
                   = loadSound(ViewState->theSound[ViewState
                                ->numSounds]);

                 updateSoundLocation
                   (ViewState->theSound[ViewState->numSounds]);
                 amplifySound
                         (ViewState->theSound[ViewState->numSounds]);
                 playSound
                   (ViewState->theSound[ViewState->numSounds]);
             }
             ViewState->numSounds++;
           break;
```

Table 18: Modifications to cmdline.C

G. SUMMARY

This implementation consists of three applications, AcousteLib, GAUDIVIEW, and GAUDIFLY. These applications, written as a family of graphical and audio modeling tools, effectively prove the concept of this thesis. They allow a modeler to easily build a graphical and acoustical model simultaneously and without added effort. The GAUDIFLY allows the user to navigate through a three-dimensional virtual world of graphics and audio. It correctly implements the generation, propagation, and reception states, although the generation state is implemented with only *wav* files.

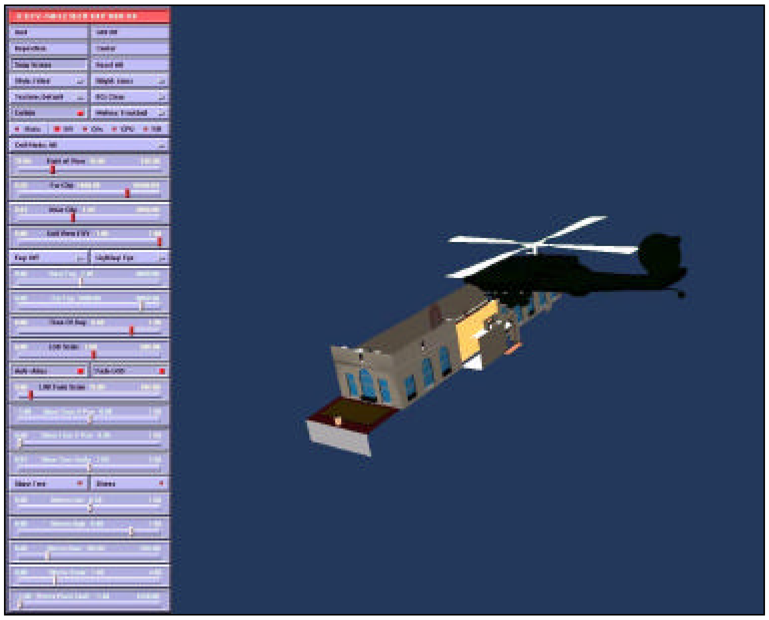


Figure 18: Screencapture of GAUDIFLY

VI. DISCUSSION

A. WHAT WOULD BE NECESSARY TO MAKE THIS A PRODUCTION MODEL?

The software developed for this thesis, AcousteLib, GAUDIVIEW, and GAUDIFLY, were for proof of concept only. Had they been developed for practical, daily use, some things would have been done differently.

1. AcousteLib

If AcousteLib were developed for daily use, four issues would need addressing. First, AcousteLib should be less hardware specific. Currently, AcousteLib runs on an SGI and uses an Acoustetron II. For more versatility, AcousteLib should be ported to the PC. It should address not only the Acoustetron II, but also all Microsoft Windows compatible sound cards.

Second, AcousteLib is currently limited to implementing only a few acoustical textures applied to walls. This is due to the hardware. If AcousteLib would maintain a list of objects within a given room, it could calculate the room’s reverberation time. The room’s reverberation time could be simulated with delays and equalization. This would allow any textures to be implemented.

Third, sound generation is another area that would need enhancing to make this product more robust. MIDI was a feature painfully missing from AcousteLib, along with other methods of sound generation like FM synthesis or sound modeling. The AcousteLib used sampled *wav* files to generate sounds; this method was elected since this thesis did not focus on sound generation, but instead focused on environment specification.

The fourth and final major weakness of AcousteLib is its neglect of ambient sounds. Ambient sounds are sounds that are not localized. The general noise level heard in a room, on a street, or in the countryside are all sounds that should be present. AcousteLib does not address sounds that are non-directional.

2. GAUDIVIEW

If *GAUDIVIEW* were developed for daily use, four issues would need addressing. The first issue concerns the viewpoint/hearpoint. Currently, the user has a birds-eye view of the model. A head represents where the sounds are being heard, but the user can pan left or right or move closer or farther while keeping the same hearing point. The head in the model could be thought of as a microphone, and the model builder had the ability to fly around the model. This was done by design. An option that would have been added, had time permitted, was an ability to set a viewpoint at the hearing location. The user would see and hear exactly what the head in the model was seeing and hearing. This could have been accomplished with a camera in Inventor.

The second issue that was not addressed is attaching a sound to a rotor. The rotor's movement would be reflected in the sound's transform. With the current version of GAUDIVIEW, updates to the audio pipeline are only applied to the sound device when the user requests.

The third issue is that GAUDIVIEW does not keep track of the room's objects and the textures of those objects. This could be done easily, if it were supported in AcousteLib. Since the AcousteLib did not support this function, it was not built into GAUDIVIEW.

The forth and last issue with GAUDIVIEW is its ability to save. Currently, GAUDIVIEW can save the whole scene graph as an Inventor model. It can also save the complete audio environment. To be more functional, the user should be able to select a subgraph and be given the choice to save as any of the following:

- An Inventor model (graphics only)
- An acoustical model (acoustical geometry only)
- An Inventor model and acoustical model (graphical and acoustical geometry)
- A sound model (acoustical properties only)
- An Inventor model and a sound model (graphical model and acoustical properties)

3. GAUDIFLY

If AcousteLib were to have been developed for daily use, the main issue that needs addressing is the ability for a model to move. Due to non-interoperability between Performer and the AcousteLib, attaching sounds to a DCS was not accomplished.

Although routines could have been written to extend Performer, but that was determined to be beyond the scope of this thesis. Attaching sounds to a DCS would have allowed a tank driving by to sound like a tank driving by.

Figure 19 demonstrates how these three applications fulfill the APP-CULL-DRAW and APP-CULL-PLAY phases that were introduced on page 36.

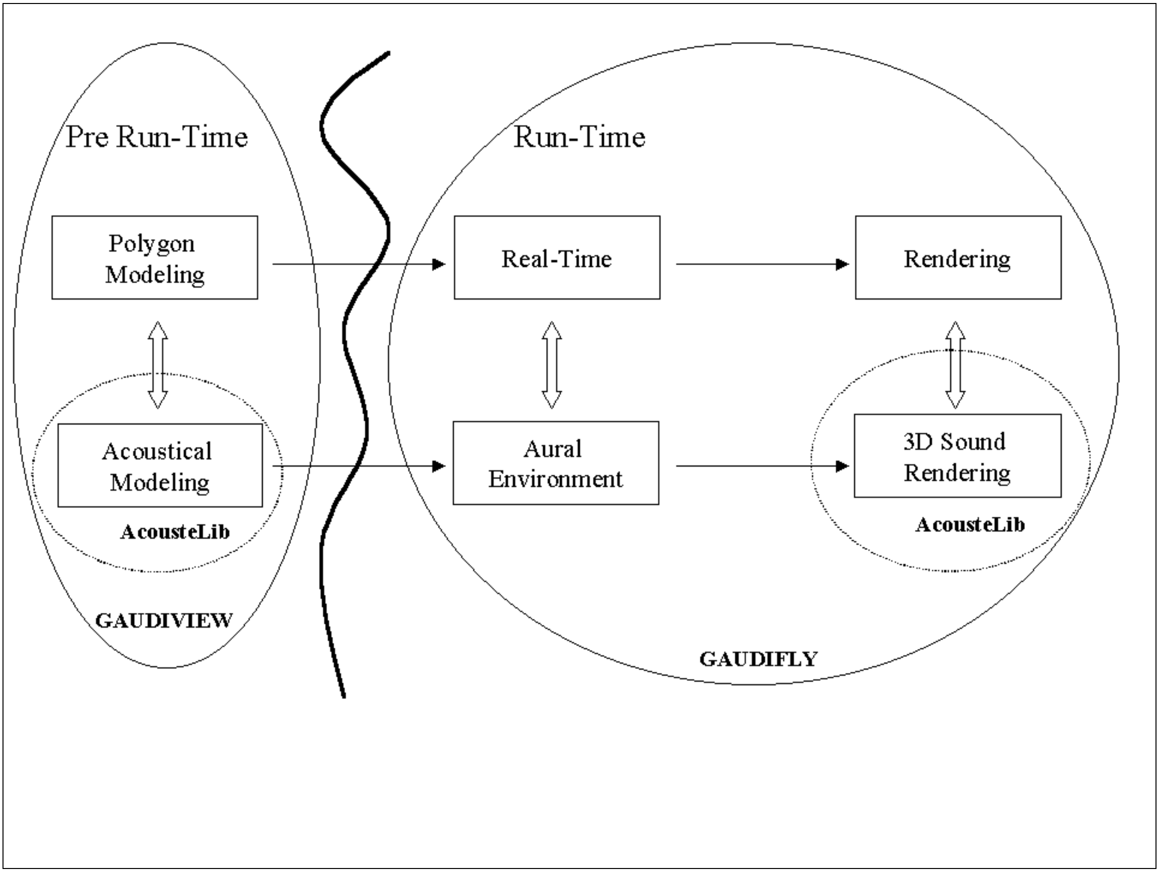


Figure 19: Resolving the applications with Figure 12

B. WHAT IS REALLY NEEDED?

1. Modelers

In modelers, the ability to create either the visual or the auditory geometry, with the other being a free by-product, is paramount. This will enable visual model builders to get the acoustical model with little or no added effort. This will also allow acoustical modelers to have a graphical representation of what they just built without necessarily concentrating on the graphical portions.

The ability to adjust parameters is also important. As a modeler sometimes must adjust the ambient properties of a polygon to make it look correct, the modeler also may need to adjust the auditory parameters from the defaults.

2. Programming

Developing a rendering engine for Performer, a browser for VRML, or any other geometry database walkthrough program will be auditory friendly when auditory models are a part of the modeling language. When a texture maps to a polygon, the acoustical properties of that texture should also map. When a polygon collides with another polygon, an auditory collision should also occur. In short, the auditory properties should be a part of, not an afterthought to, the environment.

VII. CONCLUSIONS AND RECOMMENDATIONS

After months of experiments, research, tests, and implementations, many observations about the representation of audio and the representation of graphics have been made. Many similarities between graphics and audio have been noticed. Relying upon months of observation, the following recommendations are proposed.

A. MODELING

When building models, the aural model and geometric model should be built simultaneously. Building them separately requires the builder to understand both the geometry and audio.

When a piece of geometry, say a cube, is created, it comes with default values. In Inventor, upon creation of a cube, the cube defaults to color values for red, green, and blue implemented as {0.2 0.2 0.2}. If the user wants to change the color of the cube, the user can do that. When that piece of geometry is created, it should also come with default acoustical properties. These properties could include the following:

- Timbre Tree - equates to emissive color
- Acoustical absorption coefficients - equates to reflective properties

When a piece of geometry is textured with a file, for example *wood.rgb*, an accompanying audio texture file named *wood.aud*, should accompany. The audio texture file should contain the acoustic properties of the item. Specifically, it should contain the absorption coefficients of the material that the texture file represents.

B. PROGRAMMING

All of the APIs discussed handled some of the tasks very well. None of the APIs did everything perfectly. An overarching API needs to be created. A few APIs exist, and they all do some things well. Microsoft's DirectX is very good. It contains both the geometric and the acoustic properties. Things can be attached and detached, and the audio and geometry work well together. It represents sounds well.

Aureal's A3D better represents the acoustical environment. Unfortunately, it is not part of an integrated geometry/audio API. An overarching API needs to be created. A few APIs exist, and they all do some things well. Microsoft's DirectX is very good. It contains both the geometric and the acoustic properties. Things can be attached and detached, and the audio and geometry work well together. It represents sounds well. Aureal's A3D better represents the acoustical environment; unfortunately, it is not part of an integrated geometry/audio API.

Microsoft's DirectX, or more specifically DirectSound API, is a recommend starting point. To the API it is recommended to add representations for the acoustical properties of an object - i.e., absorption coefficients. To an object, it is recommended to add a method of making its sound, such as a *wav* file (good for static noises, like an engine or a siren), a Timbre Tree (to create sounds like 'this object is being hit with a hammer'), or whatever is the method of choice. The API also needs a method of making a room. This could possibly be done with an audio equivalent of a culling frustum. When the audio is being rendered with a wave-tracing algorithm, the sounds can be calculated exactly. When the audio is being approximated by a method such as reverb, then simply perform the mathematics on the acoustic properties of all items within the room, then set an appropriate reverb.

If an API is developed like, or possibly extended from, the DirectX API, it should already work for any PC sound card that supports DirectX. PC sound cards that do not support DirectX should not be addressed, because they will not be around very long. Implementing the API for SGIs and other Unix/Linux boxes is also a strong idea. Although the lower-end markets tend to purchase Windows based systems, a large amount of innovation occurs on Unix/Linux boxes. By having a method of representing everything easily on all platforms, this method has a much better chance of taking hold.

C. NODES/DATA TYPES

Many geometry specifications exist, all with strengths and weaknesses. For this portion of the thesis, VRML will be the modeling language used. The concepts, however, apply to Inventor, Performer, Fahrenheit, and any other competitive toolkit.

A data type should be developed for acoustical absorption properties. That data type should be a vector of six floats. A sample acoustical data type is shown in Table 19.

```
Typedef struct {
      float Coef128;  // acoustical coeffecient
      float Coef256;  // acoustical coeffecient
      float Coef512;  // acoustical coeffecient
      float Coef1024; // acoustical coeffecient
      float Coef2048; // acoustical coeffecient
      float Coef4096; // acoustical coeffecient
} SFAcous;
```

Table 19: Proposed Acoustical Data Type

VRML contains a texture node called the image texture. It addresses only the graphical representation of a polygon's texture. By adding acoustical properties, the image texture would represent both the graphical and the acoustical reflection and absorption properties in the environment. A possible implementation is shown in Table 20.

```
ImageTexture {
  ExposedField MFString url     []
  Field        SFBool   repeatS TRUE
  Field        SFBool   repeatT TRUE
  Field        SFAcous  acousticCoefs
}
```

Table 20: Proposed Texture Node

The VRML sound node is the method of representing sound in VRML. The sound node can reference an audio clip. It currently allows for sound dispersion via MinBack, MinFront, MaxBack, and MaxFront fields. These fields could be default fields, but a method of representing the dispersion pattern with a structure such as the IndexedFaceSet would be appropriate for better sound modeling. This would enable the dispersion pattern of a loudspeaker to be precisely emulated. For acoustical representations that do not require as much precision, the ellipses would suffice.

VRML currently has an audio node. This node specifies an audio clip, a *wav* or MIDI file. The audio node can be played by the sound node. The audio node could be overloaded to allow for other methods of sound creation, such as Timbre Trees or PAM. By allowing a sound node to activate a sound represented by Timbre Trees or PAM, the modeler could represent one object as a bell and another object as a hammer. When these objects collided, the environment could realistically create the sound of a hammer striking a bell using the audio model.

When a polygon is instantiated in VRML, it can be given graphical properties such as textures, materials, etc. The object should also be able to have audio properties. An example of a ceiling covered in celotex is shown in Table 21.

```
Transform {
  translation -2.4 .2 1
  rotation     0 1 1  .9
  children [
    Shape {
      geometry Box {}
      appearance Appearance {
        material Material { diffuseColor 0 0 1 }# Blue
        texture ImageTexture {
                     url “celotex.jpg”
                     acousticCoefs 0.41 0.48 0.68
                                   0.79 0.75 0.55
                }
      }
    }
  ]
}
```

Table 21: Sample code for a ceiling made of celotex

Finally, neither VRML nor other modeling languages contains a specification for a room. Although they all contain a structure equivalent to a cube, this is not sufficient. A room can be made more than four walls, one ceiling, and one floor. Again, the IndexedFaceSet could be used to represent a room when an existing polygon does not work.

```
#VRML V2.0 utf8

Shape {
    Room IndexedFaceSet {
        coordIndex [ 0, 1, 3, -1, 0, 2, 3, -1 ]
        coord Coordinate {
            point [ 0 0 0, 1 0 0, 1 0 -1, 0.5 1 0 ]
        }
        color Color {
            color [ 0.2 0.7 0.8, 0.5 0 0,
                    0.1 0.8 0.1, 0 0 0.7 ]
        }
        normal Normal {
            vector [ 0 0 1, 0 0 1, 0 0 1, 0 0 1 ]
        }
        texCoord TextureCoordinate {
            point [ 0 0, 1 0, 1 0.4, 1 1 ]
        }
    }
    appearance Appearance {
        material Material { transparency 0.5 }
        texture  PixelTexture {
            image 2 2 1 0xFF 0x80 0x80 0xFF
        }
    }
    acoustics Acoustics { 0.0 0.0 0.0 0.0 0.0 0.0}

}
```

Table 22: Sample code for a Room

D. FUTURE WORK

1. BUILD A GEOMETRY/AUDIO INTEGRATED MODELER IN JAVA

Java is a write once, run everywhere programming language. A programmer astute in Java could write a modeler that could run on any platform. The geometric modeling portion of this would be academic. The audio portion would not be as easy. Currently, Java supports only the simple playing of sound file, not the ability to process the sound file in real time. Because processing sound usually requires accessing hardware - or in the case of Microsoft® Windows, accessing the Windows Foundations Class - sound processing requires writing machine specific code. Until Java develops more robust audio processing, this portion of the modeler will be difficult. Better yet, develop the audio portion of the Java programming language.

2. IMPLEMENT THE AUDIO ENVIRONMENT IN VRML

If this method were to become the best method of implementing rich acoustical environments, it means nothing unless it is implemented in a manner that is used by many people. VRML has the potential to become as prevalent in VR as the HyperText Markup Language (HTML) has become in the hypertext, or world wide web (WWW), world. By linking up with the VRML world, we can possibly affect the VR world with a lasting impact.

The VRML world has not addressed the audio environment yet, because they wanted to wait until they could dedicate the amount of time/work necessary to do it correctly. Bravo! The VRML Consortium (VRMLC) will soon begin working on developing audio. This is a golden opportunity for someone with a little ambition and a desire to make waves in a part of the VE world where it will really matter.

A first recommendation is to fight for and implement the acoustical environment parameters. Next, make the sound ellipses programmable. This can be accomplished with a dispersion formula. When a sound occurs, the loudness of that sound decreases as the distance from the sound to the listener increases. VRML has chosen to implement this phenomenon with ellipses. Although this does work, it has major problems. First, in the real world, the dispersion pattern is not eliptical. An obvious example is a loudspeaker cabinet, which usually comes with a documented dispersion pattern. Second, if a user is on one side of a wall and a sound is on another side of the wall, the user can still hear the sound while within the ellipses. The acoustical properties of the wall do not affect the hearing of the sound. Productive work exists for making the ellipses programmable, so that if a loudspeaker directs sound forward, as opposed to omnidirectional sound, the virtual environment can be represented more accurately.

3. MORE PRECISE IMPLEMENTATIONS OF SOUNDS

There are many other parallels that can be drawn between the geometry world and the audio world. These need to be worked at some time in the future. Two such areas where more precise implementation of sounds are needed include *level of detail* and *tight/loose reverb*.

Level of detail (LOD) - when a user is standing a few feet from a building, the reflections of an explosion off of the building need to be calculated. There should be a reflection from the glass, another from the eaves, etc. If the user is standing 40 meters from that building, the LOD can be much less stringent. Perhaps a single reflection from a *point source*, the building, would suffice. This could reduce the workload from the DSP - much like levels of detail reduce the workload in the graphics pipeline.

Use tight/loose reverb for smaller systems - Creative Labs has shown that tight/loose reverb can be somewhat effective for simulating room acoustics. A recommendation is to implement a formula to determine the acoustical properties of the room from the acoustical properties of each item in the room. After this, the appropriate reverb times can be calculated for each band and subsequently rendered for each band.

LIST OF REFERENCES

[ACOU96] Crystal River Engineering Inc., *Acoustetron II, The Audio RealityTM Sound Server Manual*, Crystal River Engineering Inc., Palo Alto, CA, 1996

[ALBE97] Albers, Mike, “Psychoacoustics”. [http://www.isye.gatech.edu/chmsr/Mike_Albers/projects/Varese/Psychoacoustics.html]. May, 1997

[ARON92] Arons, B. (1992). A review of the cocktail party effect. Journal of the American Voice I/O Society, 12(July)

[AURE98] Aureal Semiconductor, “3D Audio Primer”, [http://www.aureal.com/tech/primer.html], August 1998

[AURE98B] Aureal Semiconductor, “ Aureal Announces *Vortex 2*: Next Generation PCI Audio Processor ”, [http://www.aureal.com/press/1998/080698-Vortex2.htm], August 1998

[AURE98C] Aureal Semiconductor, “ A3D 2.0 Technical Brief”. [http://www.aureal.com/tech/A3D2_0tech.html]. August 1998

[BEAC97] Beacham, Frank (1997). “Sound Design For The Interactive Era”, *Pro Audio Review*, September 1997

[BEGA94] Begault, Durand, (1994). *3-D Sound For Virtual Reality and Multimedia*. Academic Press, Inc.

[BIGG96] Biggs, Lloyd, *Headphone-Delivered Three-dimensional Sound in NPSNET*, Master’s Thesis, Naval Postgraduate School, Monterey, California, September 1996

[DEUT80] Deutsch, D. (1980). “The processing of structured and unstructured tonal sequences”. *Perception and Psychophysics*, 28(5), pp. 381-389

[DEWA77] Dewar, K.M., Cuddy, L.L. & Mewhort, D.J. (1977). “Recognition of single tones with and without context.” *Journal of Experimental Psychology: Human Learning and Memory*, 3(1), pp. 60-67

[DIVA98] Takala, Tapio, *Marienkirche – A visual and aural demonstration film*, [Video Cassette] Producer: Tapio Takala, Helsinki University of Technology, July, 1998

[FERG74] Maynard Ferguson, “Gospel John”, *Chameleon*, [Compact Disc] CBS SBP 234558

[FOLE90] Foley, J.D., van Dam, A., Feiner, S.F., Hughes, J.F. (1997). *Computer Graphics, Principles and Practices* (2nd ed. in C). New York: Addison-Wesley

[FOST98] Foster, Scott. Personal Communication, January, 1998

[FRAI82] Fraisse, P. (1982). "Rhythm and tempo". In D. Deutsch (Ed.), *The psychology of music*, pp. 149-180. San Diego, CA.: Academic Press

[FOUA97] Fouad, Hesham, *Scheduling Algorithms for Real-time Sound Generation in Virtual Environments*, Ph.D. Dissertation, The George Washington University, September 1997

[GELF81] Gelfand, S.A. (1981). *Hearing: An introduction to psychological and physociological acoustics*. New York: Marcel Dekker Inc

[HAMM98] Hamm, Russell, "Significance of Musical Harmonics." [http://www.giltronics.com/rhamm.htm]. March 1998

[IRIS96] Silicon Graphics, Incorporated (1996). "Iris InSight Release 2.3.3"

[KNUD50] Knudsen, V. and Harris, C. (1950). *Acoustical Designing in Architecture*. p. 139 New York: John Wiley & Sons, Inc. London: Chapman & Hall, Ltd.

[LAWS98] Lawson, John, *Level Of Presence Or Engagement In One Experiance As A Function Of Disengagement From A Concurrent Experience*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1998

[RANE98A] Rane Corporation, " AD 22 and AD 22B Audio Delays", [http://www.rane.com/ad22.htm], August 1998

[RANE98B] Rane Corporation "GE 60 Graphic Equalizer", [http://www.rane.com/ge60.htm], August 1998

[ROSS95] Rossing, Thomas D. (1990). *The Science of Sound*. New York: Addison-Wesley

[SBLI98A] Creative Labs®, "Sound Blaster Live! Technical Specs & System Requirements" [http://www.sblive.com/product/specs.html]. August 1998

[SBLI98B] Creative Labs®, "Sound Blaster Live! Features & Benefits" [http://www.sblive.com/product/benefits.html]. August 1998

[SBLI98C] Creative Labs®, "Evolution of EAX". [http://www.soundblaster.com/eaudio/whitepaper/evolve-eax.html]. August 1998

[STOR98] Storms, Russell, *Auditory-Visual Cross-Modal Perception Phenomena*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, September 1998

[STOR95] Storms, Russell, *NPSNET-3D Sound Server: An Effective Use of the Auditory Channel*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995

[WELC98A] Welch, Norma, "Basic Acoustics and Psychoacoustics." [http://www.music.mcgill.ca/auditory/physics.html]. March 1998

[WELC98B] Welch, Norma, "Selected Moments from History". [http://www.music.mcgill.ca/auditory/history.html]. March 1998

[WERN94] Wernecke, Josie (1994). *The Inventor Toolmaker*. New York: Addison-Wesley

[WERN93] Wernecke, Josie (1993). *The Inventor Mentor*. New York: Addison-Wesley

[WOUD97] Woudenberg, Eric, " Masking and Perceptual Coding." [http://www.hip.atr.co.jp/~eaw/minidisc/MaskingPaper.html]. June, 1998

BIBLIOGRAPHY

Albers, Mike, “Psychoacoustics”. [http://www.isye.gatech.edu/chmsr/Mike_Albers/projects/Varese/Psychoacoustics.html]. May, 1997

Arons, B. (1992). A review of the cocktail party effect. Journal of the American Voice I/O Society, 12(July)

Aureal Semiconductor, “3D Audio Primer”, [http://www.aureal.com/tech/primer.html], August 1998

Begault, Durand, (1994). *3-D Sound For Virtual Reality and Multimedia*. Academic Press, Inc.

Biggs, Lloyd, *Headphone-Delivered Three-dimensional Sound in NPSNET*, Master’s Thesis, Naval Postgraduate School, Monterey, California, September 1996

Beacham, Frank (1997). “Sound Design For The Interactive Era”, *Pro Audio Review*, September 1997

Crystal River Engineering Inc., *Acoustetron II, The Audio RealityTM Sound Server Manual*, Crystal River Engineering Inc., Palo Alto, CA, 1996

Deutsch, D. (1980). “The processing of structured and unstructured tonal sequences”. *Perception and Psychophysics,* 28(5)

Dewar, K.M., Cuddy, L.L. & Mewhort, D.J. (1977). “Recognition of single tones with and without context.” *Journal of Experimental Psychology: Human Learning and Memory*, 3(1)

Ferguson, Maynard, “Gospel John”, *Chameleon*, [Compact Disc] CBS SBP 234558

Fraisse, P. (1982). Rhythm and tempo. In D. Deutsch (Ed.), *The psychology of music*. San Diego, CA.: Academic Press

Foley, J.D., van Dam, A., Feiner, S.F., Hughes, J.F. (1997). *Computer Graphics, Principles and Practices* (2nd ed. in C). New York: Addison-Wesley

Fouad, Hesham, *Scheduling Algorithms for Real-time Sound Generation in Virtual Environments*, Ph.D. Dissertation, The George Washington University, September 1997

Gelfand, S.A. (1981). *Hearing: An introduction to psychological and physociological acoustics*. New York: Marcel Dekker Inc

Hamm, Russell, “Significance of Musical Harmonics.” [http://www.giltronics.com/rhamm.htm]. March 1998

Kilgard, Mark, (1997). OpenGL™ Programming for the X Window System. New York: Addison-Wesley Developers Press

Knudsen, V. and Harris, C. (1950). *Acoustical Designing in Architecture.* New York: John Wiley & Sons, Inc. London: Chapman & Hall, Ltd.

Lawson, John, Level Of Presence Or Engagement In One Experiance As A Function Of Disengagement From A Concurrent Experience, Master’s Thesis, Naval Postgraduate School, Monterey, California, September 1998

Rane Corporation, “AD 22 and AD 22B Audio Delays”, [http://www.rane.com/ad22.htm], August 1998

Rane Corporation “GE 60 Graphic Equalizer”, [http://www.rane.com/ge60.htm], August 1998

Rane Corporation, “Rane Professional Audio Reference”. [http://www.rane.com/digi-dic.htm], August 1998

Rossing, Thomas D. (1990). *The Science of Sound*. New York: Addison-Wesley.
Silicon Graphics, Incorporated (1996). “Iris InSight Release 2.3.3”

Storms, Russell, *Auditory-Visual Cross-Modal Perception Phenomena*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, California, September 1998

Storms, Russell, *NPSNET-3D Sound Server: An Effective Use of the Auditory Channel*, Master’s Thesis, Naval Postgraduate School, Monterey, California, September 1995

Takala, Tapio, *Marienkirche – A visual and aural demonstration film,* [Video Cassette] Producer: Tapio Takala, Helsinki University of Technology, July, 1998

Takala, T., Hahn, J., Gritz, L., Geigel, J., and Lee, J.W., *Using Physically Based Models and Genetic Algorithms for Functional Composition of Sound Signals, Synchronized to Animated Motion,* International Computer Music Conference, September 1993

Welch, Norma, “Basic Acoustics and Psychoacoustics.” [http://www.music.mcgill.ca/auditory/physics.html]. March 1998

Welch, Norma, “Selected Moments from History”. [http://www.music.mcgill.ca/auditory/history.html]. March 1998

Wernecke, Josie (1994). *The Inventor Toolmaker*. New York: Addison-Wesley

Wernecke, Josie (1993). *The Inventor Mentor*. New York: Addison-Wesley

Woudenberg, Eric, "Masking and Perceptual Coding."
[http://www.hip.atr.co.jp/~eaw/minidisc/MaskingPaper.html]. June, 1998

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.. 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library.. 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Director, Training and Eductaion... 1
   MCCDC, Code C46
   1019 Elliot Road
   Quantico, VA 22134-5027

4. Director, Marine Corps Research Center.. 2
   MCCDC, Code C40RC
   2040 Broadway Street
   Quantico, VA 22134-5107

5. Director, Studies and Analysis Division.. 1
   MCCDC, Code C45
   3300 Russell Road
   Quantico, VA 22134-5130

6. Marine Corps Representative.. 1
   Naval Postgraduate School
   Code 037, Bldg. 234, HA-220
   699 Dyer Road
   Monterey, CA 93940

7. Marine Corps Tactical Systems Support Activity.. 1
   Technical Advisory Branch
   Attn: Maj J. C. Cummiskey
   Box 555171
   Camp Pendleton, CA 92055-5080

8. Dr. Rudy Darken 2
   Computer Science Dept., Code CS/DR
   Naval Postgraduate School
   Monterey, CA 93943

9. Dr. Mike Zyda 1
Computer Science Dept., Code CS/ZK
Naval Postgraduate School
Monterey, CA 93943

10. Mike Macedonia 1
Chief Scientist and Technical Director
US Army STRICOM
12350 Research Parkway
Orlando, FL 32826-3276

11. Director 1
Office of Science & Innovation
OSI, MCCDC
3300 Russell Road
Quantico, VA 22134-5021

12. Dr. Russell Storms 1
404 Cana of Galilee CourtOffice of Science & Innovation
OSI, MCCDC
3300 Russell Road
Quantico, VA 22134-5021

13. Dr. Durand R. Begault 1
Mail Stop 262-2, Room 130
NASA Ames Research Center
Moffett Field, California 94035-1000 USA

14. Mr. Scott Foster 1
Aureal Semiconductor
4245 Technology Drive
Fremont, CA 94538

15. Mr. Vince Hux 1
McCune Audio Video Lighting
222 Ramona Street, Suite 1
Monterey, CA 93940

16. Dr. Dan Boger 1
Computer Science Dept., Code CS/DB
Naval Postgraduate School
Monterey, CA 93943